TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

VILLE JUVEN

LIGHTWEIGHT EVENT-DRIVEN REAL-TIME OPERATING SYSTEM FOR RESOURCE CONSTRAINED CONNECTIVITY

Master of Science Thesis

Examiners: Prof. Timo D. Hämäläinen,
D.Sc. Teemu Laukkarinen
Examiner and topic approved on 29 March 2017

ABSTRACT

Ville Juven: Lightweight Event-driven Real Time Operating System for Resource Constrained Connectivity
Tampere University of technology
Master of Science Thesis, 62 pages
March 2017
Master's Degree Programme in Information Technology
Major: Embedded Systems
Examiner: Professor Timo D. Hämäläinen, D.Sc. Teemu Laukkarinen

Keywords: operating systems, low power, wireless connectivity

Wirepas Connectivity (WPC) is a complex protocol stack for large scale mesh-based Internet of Things (IoT) networks. The communication units in a WPC network are called nodes and these are designed to be cheap, resource constrained and battery operated. In contrast, each node requires several levels of parallel and real-time processing, which is best provided by a Real-Time Operating System (RTOS).

The resource constraint aspect places requirements for the RTOS design. The RTOS kernel should take less than 10 kB of program memory and under 1 kB of data memory. It must be energy efficient for battery operation and for this reason its scheduling must be tickless (as opposed to time-sharing). Furthermore, the WPC protocol stack requires deterministic real-time timings with microsecond accuracy from the RTOS.

This thesis studies the feasibility of related RTOSs Contiki, TinyOS, µC/OS and FreeRTOS for WPC use. The study shows that none of the related RTOSs are feasible without major modification. Contiki and TinyOS would complicate software development. µC/OS is commercially licensed and would increase per node cost. FreeRTOS lacks sufficient real-time operation for WPC. Furthermore, these RTOSs are designed to be general purpose and thus they are wasteful with precious memory and energy resources. To better deal with these challenges, a more specific approach is required.

As a solution, this thesis presents a completely new RTOS called WPC-OS, designed specifically for WPC. The RTOS design targets to timing determinism and energy efficiency in all its functions. The WPC-OS scheduler provides a novel and lightweight timetabled scheduling approach, which uses task durations to determine the next task. Event-driven operation is provided on top of this to achieve reactiveness to concurrent events.

For evaluation and measuring WPC-OS design efficiency, it was implemented on an nRF52832 platform. The measurement results show that the WPC-OS kernel achieved a small memory footprint. With the typical WPC node configuration, it uses only 5 kB of program memory and 350 B of data memory. It can handle the WPC timing requirements with its real-time event service, which guarantees 1 us timing accuracy. It provides lightweight multitasking capability for applications, while being energy efficient. WPC-OS solves all design requirements WPC imposes on RTOS design, and is suitable for mass production. As future work, coroutine and hybrid scheduling options for WPC-OS should be investigated.

TIIVISTELMÄ

Ville Juven: Vähäresurssisten toisiinsa yhdistettävien laitteiden käyttöjärjestelmäydin
Tampereen teknillinen yliopisto
Diplomityö, 62 sivua
Maaliskuu 2017
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Sulautetut järjestelmät
Tarkastaja: Professori Timo D. Hämäläinen, TkT. Teemu Laukkarinen

Avainsanat: käyttöjärjestelmät, matalavirtaisuus, langattomat yhteydet

Wirepas Connectivity (WPC) on kompleksinen protokollapino suuren mittaluokan langattomia mesh-verkkoja varten. Laitteet WPC verkossa ovat halpoja, vähäresurssisia ja paristokäyttöisiä. Tästä huolimatta jokainen laite vaatii tavan suorittaa useantasoista rinnakkaista ja reaaliaikaista laskentaa. Reaaliaikainen käyttöjärjestelmä (RTOS) soveltuu näiden vaatimusten täyttämiseen parhaiten.

Muistin ja laskentatehon puute asettaa vaatimuksia RTOS:n suunnitteluun. RTOS ytimen tulisi viedä alle 10 kilotavua ohjelmamuistia ja alle 1 kilotavua datamuistia. Sen tulee olla energiatehokas, ja tästä syystä ohjelmien ajastus ei saa perustua aikajakoon. Näiden lisäksi WPC protokollapino vaatii RTOS ytimeltä mikrosekunnin tarkkoja ajoituksia, missä ei saa olla hajontaa.

Tämä työ tutkii olemassa olevien, potentiaalisien RTOS ytimien soveltuvuutta WPC käyttöön. Tutkimus osoittaa että olemassaolevat ratkaisut eivät sovellu tähän ilman isoja muutoksia. Contiki ja TinyOS ovat liian monimutkaisia ja täten vaikeuttaisi sovelluskehitystä. µC/OS on kaupallinen ratkaisu ja nostaisi laitekohtaisia hintoja. FreeRTOS ei tarjoa tarpeeksi tarkkoja ajastuksia WPC:lle. Nämä ytimet ovat suunniteltu yleiskäyttöisiksi, ja täten niiden muistin- ja energiankulutus ei ole optimoitua. Näiden haasteiden ratkaisemiseksi tarvitaan siis selvästi enemmän kohdennettu lähestymistapa.

Ratkaisuna näihin ongelmiin tässä diplomityössä esitellään täysin uusi RTOS ydin, nimeltä WPC-OS. Se on suunniteltu vastaamaan nimenomaan WPC:n tarpeisiin. Sen suunnittelussa on painotettu ajoitusten determinismiä ja energiatehokkuutta. Se tarjoaa kevyen tavan ajaa ajastettuja tehtäviä, sekä reaktiivisen, tapahtumapohjaisen tavan ajaa rinnakkasia tehtäviä.

Evaluointia ja mittauksia varten WPC-OS toteutettiin nRF52832 alustalle. Mittaukset osoittavat että WPC-OS:n ydin käyttää muistia tehokkaasti. Tyypilliselle WPC laitteelle konfiguroituna se käyttää ohjelmamuistia vain 5 kilotavua ja datamuistia 350 tavua. WPC-OS on energiatehokas ja tarjoaa kevyen moniajoytimen. Sen reaaliaikaiset tapahtumat takaavat 1:n mikrosekunnin ajoitustarkkuuden. Tämä vastaa WPC:n ajoitusvaatimukseen. WPS-OS ratkaisee kaikki WPC:n RTOS ytimelle asettamat vaatimukset. Jatkokehitysideana voisi tutkia hybridiskeduloinnin toteutusta WPC-OS ytimeen.

PREFACE

This thesis was done for Wirepas at the Wirepas Tampere office premises. The design and implementation was completed in the fall of 2014 and writing the thesis begun in March 2017.

I would like to express my sincere gratitude to Wirepas for allowing me to use office hours for writing the thesis. The never-ending support and understanding I received will never be forgotten. I would also like to show gratitude to my thesis supervisors Prof. Timo D. Hämäläinen, D.Sc. Teemu Laukkarinen for their support and invaluable guidance during the writing process of this thesis.

I would also like to thank my colleagues, D.Sc. Ville Kaseva, M.Sc. Kari Pihl and M.Sc. Pedro Silva for providing me with insightful comments and assistance with the written thesis.

Finally, I want to thank my family and friends for understanding and mental support during the writing of this thesis.

Tampere, 16.5.2017

Ville Juven

CONTENTS

1. INTRODUCTION 1
1.1 Thesis motivation and scope 3
1.2 Thesis outline 4
2. OPERATING SYSTEMS 5
2.1 Operating system responsibilities 5
2.2 Real-Time Operating Systems 6
2.2.1 Scheduler 8
2.2.2 Tasks 12
2.2.3 Synchronization 13
2.2.4 Memory management 14
2.2.5 Interrupts 15
3. RELATED REAL-TIME OPERATING SYSTEMS 16
3.1 Contiki 16
3.2 TinyOS 16
3.3 FreeRTOS 17
3.4 Micro-Controller Operating Systems (µC/OS-II/-III) 18
3.5 Comparison 18
3.6 Conclusions 19
4. MOTIVATION AND REQUIREMENTS FOR WPC-OS 20
4.1 Wirepas Connectivity 20
4.2 General requirements for the kernel 23
4.3 Timing requirements for the kernel 25
4.4 Scheduling options for WPC 25
4.5 Preemptive scheduling for WPC 26
4.6 Event-driven (non-preemptive) scheduling for WPC 26
4.7 Resource constraints for WPC 26
5. WPC-OS DESIGN 29
5.1 Design and architecture 29
5.2 Tasks 30
5.3 Scheduler 31
5.4 Services 32
5.4.1 Events 32
5.4.2 Timers 33
5.4.3 Message queues 33
5.4.4 Drivers 33
5.4.5 Memory Management 34
5.5 Interrupts 35
6. WPC-OS IMPLEMENTATION 36
6.1 Programming language and compiler 36

6.2 Kernel 36
6.2.1 Tasks 36
6.2.2 Scheduler 38
6.2.3 Events 38
6.3 Timers 39
6.3.1 Timer HAL 39
6.3.2 Timeouts 40
6.3.3 Real-time service 41
6.4 Energy management 42
6.5 Message queues 42
6.6 Interrupt priorities 44
6.7 System calls 44
6.8 List of files and directories 47
7. EVALUATION AND MEASUREMENTS 48
7.1 Measurement setup 48
7.2 Real-time events 49
7.3 Energy efficiency and energy management 49
7.3.1 Deep sleep utilization 51
7.3.2 Idle task utilization 51
7.3.3 Context switching delay 51
7.4 Memory footprint 52
7.4.1 Memory footprint comparison 53
7.5 Example task 53
7.6 Running WPC with WPC-OS 55
7.7 Evaluation and measurement results 56
8. CONCLUSIONS 57
REFERENCES 59

LIST OF FIGURES

Figure 1.WPC topology *2*
Figure 2.WPC software architecture *4*
Figure 3.Priority-based scheduling with 3 task priorities [4] *8*
Figure 4.Context switching delay after event *9*
Figure 5.Typical FSM in a priority based, preemptive scheduler [4] *10*
Figure 6.Task scheduling in a preemptive operating system [5] *10*
Figure 7.Task scheduling in an event-driven operating system [9] *11*
Figure 8.Memory fragmentation and its problem illustrated *15*
Figure 9.Superframes repeated every access cycle *20*
Figure 10.Superframe structure *21*
Figure 11.Slot boundary with TX and RX margins *22*
Figure 12.WPC-OS architecture *29*
Figure 13.Task scheduling example with priority elevation by asynchronous event *31*
Figure 14.Dynamic memory management by slab allocation *34*
Figure 15.Measuring instrument screen *48*
Figure 16.Task wake-up by real-time event *50*
Figure 17.WPC with WPC-OS test setup *55*
Figure 18.Terminal output from both nodes *56*

LIST OF TABLES

Table 1.Comparison of popular embedded RTOS features *18*
Table 2.List of supported WPC platforms *27*
Table 3.Summary of WPC-OS requirements *28*
Table 4.Example of NVIC interrupt priorities *44*
Table 5.Task control and kernel internal service calls *45*
Table 6.Operating system service calls *46*
Table 7.List of files and folders of WPC-OS implementation *47*
Table 8.Real-time event timing statistics *49*
Table 9.Measured timings from energy efficiency setup *50*
Table 10.WPC-OS memory footprint with WPC *52*
Table 11.Memory usage of WPC-OS with WPC example *56*
Table 12.Evaluation and measurement results *56*

LIST ABBREVIATIONS

AES	Advanced Encryption Standard
ARM	Advanced RISC Machine
ADC	Analog to Digital Converter
API	Application Programming Interface
CPU	Central Processing Unit
CAP	Contention Access Period
CF-MAC	Contention Free Medium Access Control
CFP	Contention Free Period
XTAL	Crystal Oscillator
FSM	Finite State Machine
FTDMA	Frequency-Time Division, Multiple Access
GPIO	General Purpose Input / Output
GPOS	General Purpose Operating System
HAL	Hardware Abstraction Layer
HFCLK	High Frequency Clock
IoT	The Internet of Things
IPC	Inter-Process Communication
ISR	Interrupt Service Routine
6LoWPAN	IPV6 over Low power Wireless Personal Area Networks
LED	Light Emitting Diode
LUT	Look Up Table
MMU	Memory Management Unit
MPU	Memory Protection Unit
MCU	Microcontroller Unit
NVIC	Nested Vectored Interrupt Controller
PRS	Peripheral Reflex System
PCB	Process Control Block
PID	Process Identifier
RTC	Real Time Clock
RTOS	Real Time Operating System
RISC	Reduced Instruction Set Computer
CBP	Cluster Beacon (Signaling) Period
SPI	Serial Peripheral Interface
SoC	System-on-Chip
SysTick	ARM System Timer
TCB	Task Control Block
TDMA	Time Division, Multiple Access
UART	Universal Asynchronous Receiver / Transmitter
WPC	Wirepas Connectivity
WPC-OS	Wirepas Connectivity Operating System
WSN	Wireless Sensor Networks

1. INTRODUCTION

The Internet of Things (IoT) [1] provides connectivity for devices such as consumer electronics, vehicles, buildings, clothing and other small appliances that are not traditionally expected to have any [40]. It means that devices can connect with each other over the Internet making IoT applications possible. These applications include environment monitoring, building- and home automation, water- and electricity metering, control systems (actuators) and asset management [31] to name a few. IoT applications are expected to make everyday life easier, for example by allowing remote electricity meter reading, instead of having to do it manually on site.

Providing connectivity to an ever increasing number of devices is a challenge with current broadband and mobile Internet technologies, which require one Internet connection per device. For example, the wireless mobile network (3G) requires a SIM card for each device [32].

Wirepas Connectivity (WPC) [8] provides a solution where the devices form the IoT network by connecting directly with each other and handle the communication collaboratively. Internet access to and from the WPC network is handled by an Internet gateway. The total cost of a device decreases due to not needing a dedicated connection for each device, for example by removing the cellular connection cost. The lifetime cost decreases by eliminating the need for maintenance labor and repeated installation.

The embedded IoT devices in a WPC network are called *nodes* and the Internet gateway is called a *sink*. The nodes form multi-hop paths towards sinks and data is forwarded from node to node until the sink is reached. Figure 1 illustrates how the network forms. The data paths form a *tree-like* routing topology over the mesh network, where the sink is always the root of the tree. A node can be simultaneously connected to multiple neighbors and multiple sinks for robustness and load balancing. In case of node failure, the nodes will repair the network topology by finding alternate paths. This increases fault tolerance as WPC does not have single points of failure and the WPC network works even if multiple nodes fail. Fault tolerance and autonomous operation eliminates the need for human interference, which consequently decreases lifetime costs.

The nodes communicate with each other in pre-determined time slots to share the wireless medium. The time slots are determined by a *Time Division, Multiple Access* (TDMA) schedule. Wireless access is performed by the WPC *Contention Free Medium Access Control* (CF-MAC) protocol.

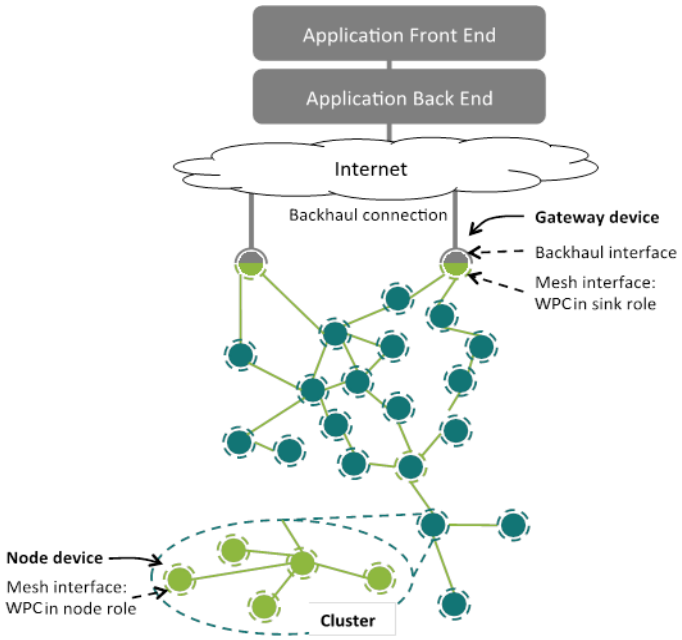

Figure 1.WPC topology

The timing critical moments are the beginnings of a TDMA slot, or *slot boundary*. Communication can only happen at these slot boundaries and only between specific nodes. This schedule results in low duty cycle operation and efficient spectrum usage communication-wise. This increases energy efficiency and removes intra-network collisions.

For the TDMA schedule to work, the communication timings must be very accurate. Timing inaccuracy is compensated by adding margins to slot boundaries which increase the communication duty cycle. Timing accuracy directly affects the margins, and more precise timing means the possibility to use tighter margins resulting in lower power consumption. As an example, 1 us timing accuracy on an nRF52832 platform [21] would result in 149 uA average current consumption. In contrast, 1 ms timing accuracy would result in 243 uA average current consumption. Converted to lifetime figures when using a standard AA battery (2000 mAh capacity), this would shorten the battery life from 19 to 11 months. Thus, the timing accuracy is crucial to WPC energy efficiency.

WPC is designed to run on resource constrained embedded platforms with low power consumption, limited processing power and memory for applications. A typical WPC node is expected to work for years with standard AA-batteries without battery changes. Therefore, a typical WPC node consists of a small Microcontroller Unit (MCU), a digital radio transceiver, and application sensors. These can be either a System-on-Chip (SoC) or discrete components. Such resource constrained hardware reduces per node cost, but the low processing performance and limited available memory complicates software design.

1.1 Thesis motivation and scope

Typically, simple embedded programs are run as *bare-metal*. A bare-metal program often performs a single task, for example reading inputs and performing actions on the embedded hardware. The program sequence executes in a single monolithic loop and the hardware access is handled in a de-centralized manner.

Non-monolithic execution and centralized hardware access become necessary when program complexity and the amount of concurrent tasks increase. An operating system can be used to achieve both. The program sequencing is handled by the operating system in a non-monolithic manner. The operating system takes ownership of the embedded hardware meaning the programs are prohibited from accessing it directly. The operating system provides centralized services for creating and running tasks, and accessing hardware [6].

A typical WPC application runs on both the nodes and a back-end system. Thus, the ability to run user application tasks together with WPC tasks on the nodes is essential. A WPC node is expected to support a multitude of user applications with different timing requirements, while participating in the WPC network. The application might require real-time operation, for example if an alarm is triggered, the alarm message must be sent at once.

The strict timing requirements of the WPC protocol stack and the need to run multiple applications motivate the use of a Real-Time Operating System (RTOS) for WPC. An RTOS ensures that user applications do not interfere with WPC timings. Further, it provides shared access to hardware for the stack and the applications.

The scope of this thesis is the RTOS design. This is illustrated in Figure 2, which presents the layered design of WPC. WPC includes the RTOS kernel, Hardware Abstraction Layers (HAL), the WPC protocol stack and an application support layer for accessing the stack and RTOS. The timing critical parts of the WPC stack will be explained, as they place requirements for the RTOS design.

This thesis studies the following requirements WPC imposes on RTOS design:

1. High precision real-time operation for CF-MAC timings
2. Energy management for preserving energy
3. Tickless operation for battery operation
4. Optimized program and data memory footprint
5. Ease of use for application development
6. Multi-tasking capability

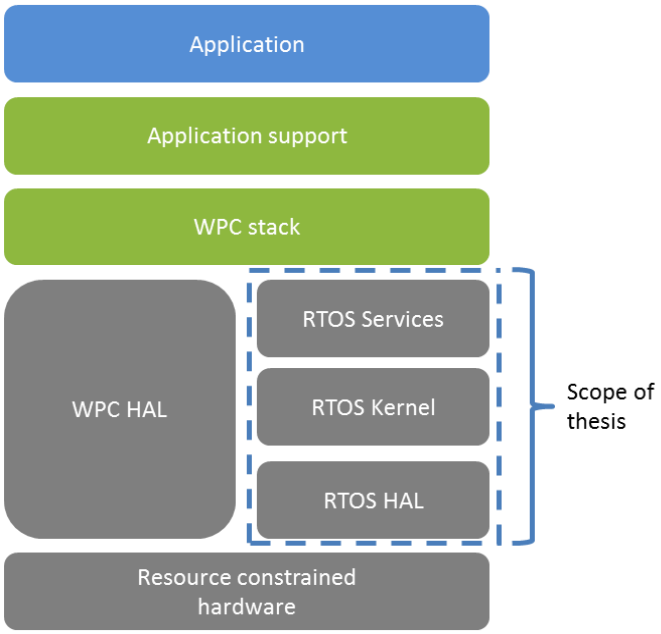

Figure 2.WPC software architecture

As a result, this thesis presents an RTOS kernel called Wirepas Connectivity Operating System (WPC-OS). WPC-OS is designed to handle WPC timing requirements, while offering multitasking capability and safe access to shared hardware for user applications. The kernel is designed to be build time configurable to account for platform resource constraints and application needs. It provides dynamic memory and inter-process communication (IPC) interfaces for easy communication between the stack and the applications. It also handles system energy management. All services are offered via a simple system call Application Programming Interface (API) that hides the semantics of the operating system itself. The RTOS design, implementation and evaluation with measurement results are presented to demonstrate WPC-OS feasibility.

1.2 Thesis outline

The rest of the thesis is organized as follows. Chapter 2 describes what an operating system is. Common operating system components and requirements are introduced. Definitions for RTOS and event-driven are provided. Chapter 3 presents commonly referred RTOSs: Contiki, TinyOS, FreeRTOS and µC/OS-II/-III. Chapter 4 focuses on motivation for WPC-OS and its requirements. WPC-OS design is presented in Chapter 5, while the implementation is in Chapter 6. Evaluation, measurements and comparison with Contiki is given in Chapter 7. Chapter 8 concludes this thesis.

2. OPERATING SYSTEMS

This chapter explains what an operating system is, what its responsibilities are and how it typically handles them. An overview of feasible operating systems for connectivity on resource constrained embedded systems is also given. Common requirements for the kernel and hardware are listed. A definition is given for an RTOS.

2.1 Operating system responsibilities

The main purpose of an operating system is to make using a computer and running user applications easier. It handles multitasking, and multi-user operation on a single hardware. It provides services for creating and running tasks, access to hardware, centralized access to files, error detection and dynamic memory management. It takes ownership of the hardware, so users do not have to implement hardware control in the applications. This is especially necessary if the hardware has multiple users; concurrent access must be abstracted and protected [33].

The operating system distributes Central Processing Unit (CPU) time to applications. Applications are called *tasks*, or *processes*. They run in their own *context*, which is a sandbox where a task resides. They are changed by an event called *context switch*, and the execution time is distributed by an entity called a *scheduler*. They do not know semantics of each other, and communication between them is handled by the operating system.

The operating system protects itself from illegal access and makes sure applications do not interfere with each-other or with the operating system itself [7]. This is achieved by separating memory areas so that user applications cannot access protected memory. This memory area separation is denoted by the terms *user space* and *kernel space* and requires hardware support, for example a Memory Management Unit (MMU) [28]. If an illegal access from user space is detected, the MMU will raise an exception that forces a context switch into kernel space. The exception is then handled by the operating system and the task can be forcibly removed from execution.

An operating system that takes complete ownership of hardware resources is called a General Purpose Operating System (GPOS), and the kernel type is a monolithic kernel. This is the typical kernel implementation in microcomputers, for example PCs and MACs with mainly human users and interactivity in mind.

Other common kernel types are microkernels and hybrid kernels. Microkernels allow applications to implement drivers, while the kernel does for example memory management and protection. In hybrid kernels some of the hardware and drivers reside in the kernel space, while the rest is freely accessible to the user [5, 7].

The monolithic approach is not necessarily feasible when designing an operating system for resource constrained systems, for example embedded systems. Resource constrained hardware might not include a MMU for memory area protection or implementing the protection might increase operating system overhead too much.

Embedded operating systems sacrifice safety for performance, which in turn complicates application design as the application must ensure itself that it behaves. Typically, embedded system kernels are microkernels or hybrid kernels, where the hardware ownership is only symbolic, as the operating system does not protect itself for illegal access and the application code can see the same hardware as the operating system does.

2.2 Real-Time Operating Systems

An RTOS is "characterized by having time as a key parameter" [7] and it can be defined as "A real-time operating system (RTOS) is a program that schedules execution in a timely manner, manages system resources, and provides a consistent foundation for developing application code" [4]. An RTOS must therefore support timed process scheduling, hardware control and access to the RTOS resources.

Access to the RTOS kernel is provided by an interface called a *system call API.* This is a collection of functions for requesting services from the RTOS. A system call must be handled in an *atomic* manner, meaning the CPU cannot be interrupted when modifying the kernel resources. The implementation of a system call API function can be *blocking* or *non-blocking*. A non-blocking system call does not halt process execution, while a blocking call suspends the process until a condition is met [4].

When a process requires services from the kernel, it sets up parameters for a system service call, and executes a special *trap* instruction, or sets a kernel event to forcibly switch context to kernel space. The kernel then figures out what the calling process wants to do by inspecting the parameters, carries out the requested operation and returns control and the result of the operation to the user space process [7].

Another way of performing a system call is to block context switching for the duration of the system call. The calling process modifies the kernel owned or shared resources directly in an atomic section. This is the only safe way of performing a system call if the hardware does not support separation between user space and kernel space.

A typical RTOS kernel implements the following components [4]:

- **A scheduler**: This handles distributing CPU time for processes, more specifically the amount of CPU time and when the process gets executed.
- **Tasks**: These are unique and isolated entities with a specific purpose (or many) and the scheduler is responsible for switching between the tasks depending on priority and events.
- **Synchronization**: In a multi-tasking environment, concurrency protection and synchronization is necessary; for example one task might need another task to finish its work before it can continue.
- **Input / Output (I/O) control**: For example handling disk access, or access to shared hardware.
- **Memory management**: Dynamic memory allocation, distribution and freeing.
- **Protection**: Memory area protection for the kernel, applications, and optionally virtualization if the hardware supports it.
- **Interrupt access**: The kernel usually takes ownership of interrupts, meaning the actual interrupt vectors are owned by the operating system and the Interrupt Service Routine (ISR) is executed by the operating system in kernel space, while the actual task triggered by the interrupt is deferred to user space.
- **Timing services**: System time and optional calendar up-keeping. Timing and execution of timed events.
- **Inter-Process Communication** (IPC): IPC messaging offers a way for the processes to communicate with each other in an unambiguous manner via a message pipe. The operating system allocates and maintains these *pipes* and *pumps* the messages between processes.

Typically, real-time systems with RTOSs are closed systems, where the system can only run trusted programs developed by the system designer and user applications are not supported. Thus, the protection aspect offered by GPOSs can be neglected in RTOS design [7].

Timing-wise the implementation and requirements of a RTOS also depends on the intended application and real-time systems are separated into two categories; *hard-*, and *soft* real-time systems. The definitions for these categories are [7]:

- **Soft real-time systems:** These are systems that have a real-time requirement but missing deadlines occasionally is not fatal. This can include multimedia systems, digital audio players and smartphones.
- **Hard real-time systems:** In these systems missing a deadline is fatal. For example assembly plants with robots performing specific timed functions are hard real-time systems. Other examples include avionics, automotive, medical, military and nuclear applications.

Hard systems require determinism from the operating system and scheduler. Hard systems are often implemented as static libraries; linked, and run tightly intertwined with the application code with no protection or segregation between the two.

With soft systems the managing of real-time events can be more relaxed and not even directly tied to task scheduling. If the execution of a task is delayed by another task, the task waiting will miss the timed operation intended and attempt it later if necessary. Typical soft systems are commercial applications designed for consumer use [7].

2.2.1 Scheduler

Real-time operating systems are different from the usual monolithic operating systems, especially in timing and scheduling functions. A traditional operating system typically targets to fairness in the scheduling, where tasks get execution time evenly. The benefit is that a task cannot be completely *starved* of CPU time. The problem is that timings are non-deterministic, as a higher priority task can be preempted by a lower priority task. In GPOSs, priorities control how much execution time is given in relation to each other [5].

Real-time operating systems implement strict priorities and the execution order is deterministic; if a higher priority task has something to do, it is always scheduled regardless of how much time it has used in the past [4]. Figure 3 illustrates how lower priority tasks get preempted when a higher priority task has work to do and are blocked until a higher priority task yields.

Scheduling can be *preemptive* or *non-preemptive* [5]. With preemptive scheduling the context is forcibly switched. With non-preemptive scheduling the task being executed always runs to completion. In either case, the status of the blocked task has to be saved, and the status of the replacing task must be loaded. Context switching is always overhead, as nothing useful from the applications point of view is happening when the operating system is saving and restoring task statuses. This is why context switching times should be kept at a minimum, to preserve CPU time.

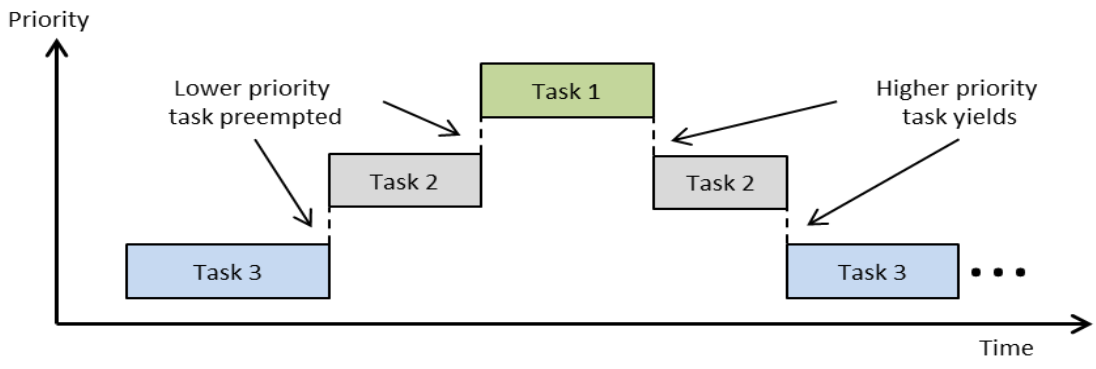

Figure 3.Priority-based scheduling with 3 task priorities [4]

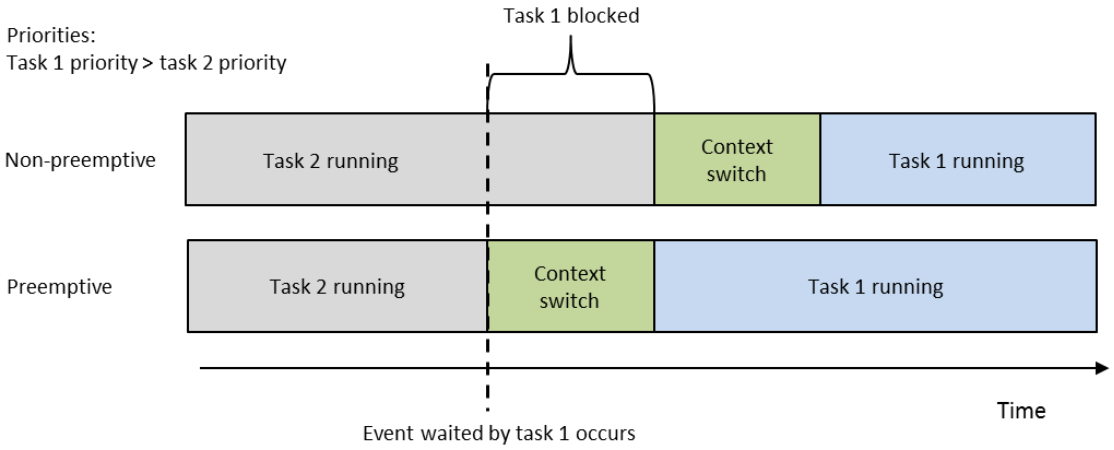

Figure 4.Context switching delay after event

In a preemptive RTOS, context switching overhead is made deterministic. Preemptive RTOSs have a guaranteed context switching time when a higher priority task indicates it has work to be done. Figure 4 illustrates how in non-preemptive kernels the context switch happens whenever the lower priority task decides to yield, but on preemptive cases the switch is done almost immediately by the kernel when an event waited by the higher priority task occurs. A context switch can happen, for example, when a timer event or an asynchronous event occurs [4].

A scheduler typically has three states for the tasks: *Ready*, *blocked* and *running*. These states are defined as:

- **Ready:** The task is ready to run, but a higher priority task is blocking it.
- **Blocked:** The task is waiting for a resource, event or has delayed itself for a discrete amount of time.
- **Running:** The task is the highest priority task and is running.

Figure 5 illustrates how these states form a Finite State Machine (FSM) the scheduler runs and how it moves tasks from one state to another.

Additionally, some kernels use more granular states, such as *suspended*, *pended*, and *delayed*. Pended and delayed are more detailed states of the blocked state, where pended means the task is waiting for a resource, and delayed means the task is waiting for a timer event. An event can change the state of a task, but might not induce a context switch as the highest priority task is unaffected. The suspended state is usually for debugging purposes and not a part of the normal operation [4].

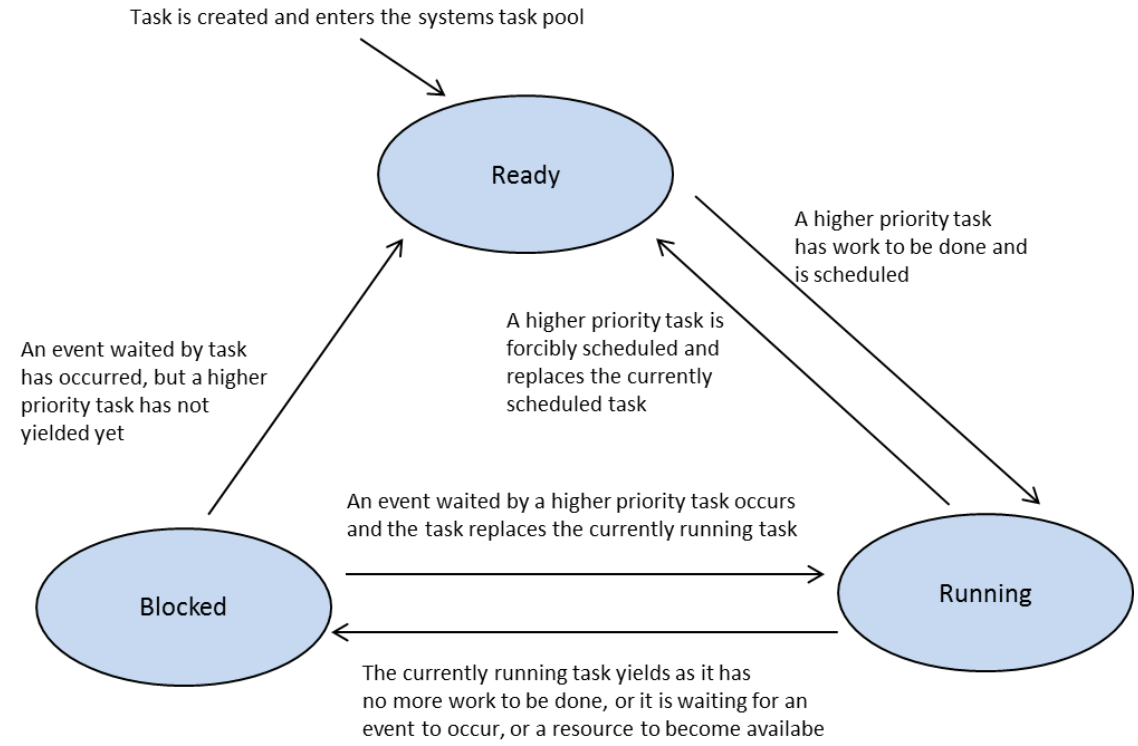

Figure 5.Typical FSM in a priority based, preemptive scheduler [4]

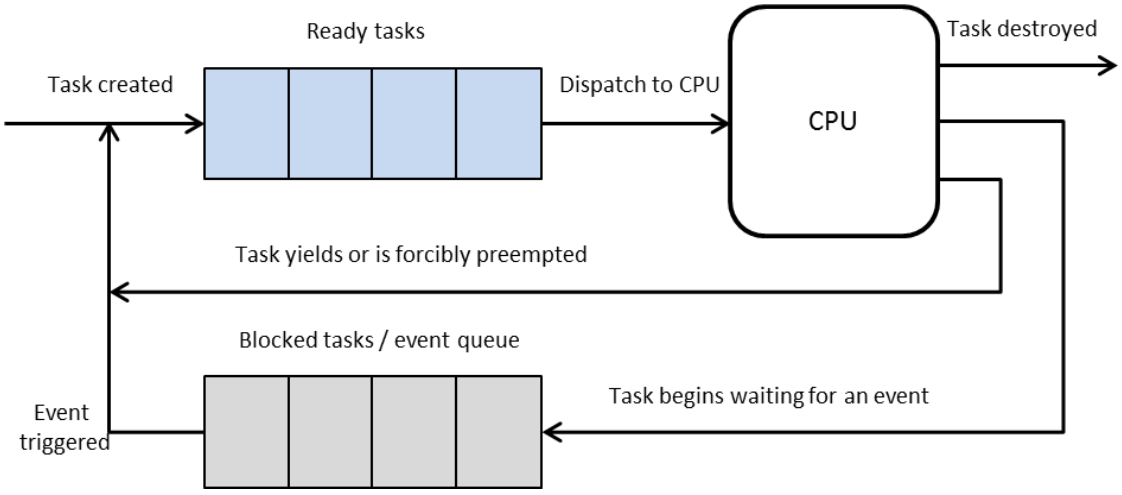

Figure 6.Task scheduling in a preemptive operating system [5]

One example implementation of a scheduler is a round-robin scheduler. The tasks reside in a round-robin queue, or queues. The scheduler might have separate queues for blocked and ready to run tasks, but at least a ready queue is necessary. Figure 6 illustrates how this round-robin scheduling shuffles the tasks using the scheduler FSM principle [5].

When a task is created it is appended in the ready queue. When a context switch occurs, the scheduler takes the first (or last depending on ordering) task from the ready queue and dispatches it to the CPU to run.

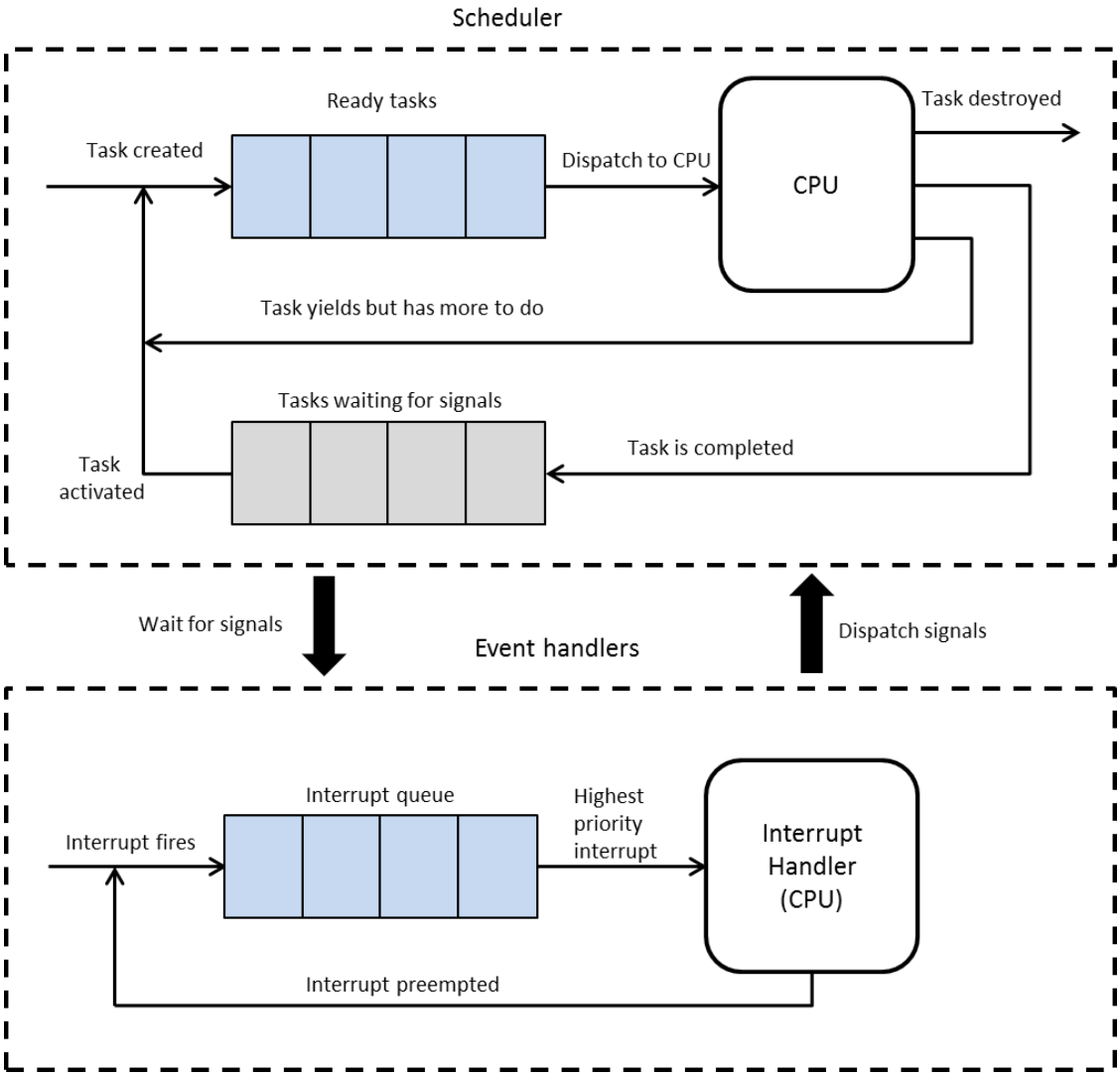

Figure 7.Task scheduling in an event-driven operating system [9]

In a non-preemptive scheduler the same basic states exist, but moving tasks from running state to other states cannot be done immediately when a higher priority task changes state. The currently running task gets to finish its work and after it yields, the scheduler moves the higher priority task to the running state. The downside of this approach is the complication of task design and distributing processing time in the most efficient manner becomes more difficult.

An event-driven scheduler runs an event handler or handlers in addition to task scheduling as illustrated in Figure 7 [9]. The events are generated by hardware interrupts, indicating, for example, that a certain time has passed. Event handlers run from interrupt context provide concurrency and reactiveness; when something important happens, for example, data reception from radio, the event handler preempts the currently running task and triggers the data reading process [10, 9]. As modern processors support several layers of interrupt priority, the event handlers have implicit priorities and one can preempt the other, providing a deeper level of concurrency [11].

Event notifications can be done by setting a signal in the tasks metadata structure, and when the scheduler is entered next time, the operating system knows an event owned by the task has occurred. These signals are set by the event handlers in interrupt context and consumed by the tasks in thread context [4].

Tasks themselves can also set event signals via the operating system for example by sending messages to other tasks or themselves. If, for example, a low priority task requires long processing times, it can do it in smaller time slices. Before yielding it can send a message to itself to inform the scheduler that it has not finished processing yet. The scheduler can then re-schedule the low priority task if there is time.

2.2.2 Tasks

A *task, or a process,* is an operating system executable that runs on shared hardware with other tasks. Sometimes tasks or processes are called *threads*. Threads are spawned and executed inside a process, thus a process always has at least one thread where the program code is executed. The difference is that a process has its own memory space, while threads inside a process share the same memory space [7]. This thesis will use the term task as a synonym to all executables, such as processes and threads, as a task requires at least one process, or thread to execute.

The main components of a task include [4]:

- **A Task Control Block (TCB), or Process Control Block (PCB):** this contains information about the task for the operating system, for example task state, context, program counter and a process identifier (PID). The TCB also contains scheduler metadata e.g. task priority and how long the process has been running.
- **Task procedure:** The sandbox where the task is executed in.
- **Stack memory:** For storing temporary variables of the procedure and sub-procedures run by the task.
- **I/O status information:** This contains all the open files, drivers and I/O connections owned by the task.

And with operating systems supporting Virtualization or a MMU:

- **Memory / virtualization information:** This contains all the reserved memory resources owned by the task.

Another task type is a *co-routine* [17] which is a cooperative, non-preemptive multitasking method. Co-routines run cooperatively only with respect to each other and can be run from within preemptive processes. Within a process, a co-routine can only get scheduled when the current co-routine yields. Cooperative operation minimizes context switching overhead as information needed by preemptive scheduling is not saved.

All co-routines within a task share a single stack, minimizing data memory usage. This however requires special attention when designing applications, as local variables are not preserved when a co-routine yields. This problem can be solved by storing the necessary variables in static variables.

Tasks are usually split into several working threads with separate responsibilities but with some dependency to each other. The kernel also spawns itself and its sub-routines as a task. As the processor cannot be stopped from executing unless put to sleep, the operating system must implement an idle task that is run whenever no other task is requesting CPU time [4].

2.2.3 Synchronization

Synchronization means waiting for something to happen before continuing, for example waiting for another task to complete something the waiting task needs, or for an event to occur. This event can be for example waiting for an I/O operation to complete, or for a timer interrupt indicating that a time has passed. Task synchronization is done by synchronization objects.

A semaphore is a synchronization object, which implements a reference count barrier for accessing shared resources. Semaphores have two operations, *wait* and *signal.* When a task performs a wait operation the reference count is decremented and the current value is tested in an atomic section. The task may proceed if the counter is greater than 0, otherwise the task enters the semaphores waiting queue. Signaling a semaphore increments the reference counter allowing the next task to continue. Any task can signal a semaphore, meaning no ownership information is stored. Semaphores also implement a maximum value for the reference count. A special case of a semaphore is a *binary semaphore* that has a maximum value of 0 (or 1) [7].

A mutual exclusion block (mutex) is a semaphore with stored owner information. Mutexes can be simplified as binary semaphores with the difference that only the current owner can unlock the mutex. A mutex ownership is *acquired* with the wait operation and *released* with the signal operation [5].

A message queue is a way to exchange information between tasks, but queues can also be used for synchronization. One task might, for example, wait for data from another task, and this data is exchanged via a message queue. A *consumer* starts waiting for data and enters a blocked state. The data *producer* is given execution priority as the other task is waiting for data from it. When the data producer is ready, it sends the data to the message queue. The operating system conveys this data while moving the consumer to ready state [4].

2.2.4 Memory management

An operating system handles two aspects of memory management: managing memory allocation for users from physical memory, and managing virtual memory or the MMU. The virtual memory management is usually a driver and control block for the physical memory. Users access the physical memory locations via virtual addresses; for example the application might see an address space beginning from 0x0000, while the actual memory is located at 0x1000. Virtual memory is not common in resource constrained embedded systems due to lacking hardware support. Virtualization also creates processing overhead when accessing memory and the needed control blocks require memory that cannot be used to store user data [7].

When a task is created, the operating system allocates memory needed to store variables and the executable instructions. This memory is statically allocated. An operating system also provides dynamic memory management, meaning tasks can request for extra memory when needed. This dynamic memory is called *heap* memory and its organization can be done in several ways. Dynamic memory management creates overhead which must be avoided in RTOS design, but viable options exist.

One option is to divide the heap into blocks and create a Look Up Table (LUT) containing the status of these blocks. When a task requests for N bytes of dynamic memory, the LUT is consulted for the best fitting continuous block of free memory. Consulting the LUT is iterative, meaning allocation times are non-deterministic [4].

The LUT based approach suffers from memory fragmentation, meaning there is no guarantee that the remaining free memory is continuous. This occurs when the memory is filled with small allocations that are not freed in the same order, resulting in gaps of free memory blocks between allocated blocks. The total amount of free memory no longer reflects the maximum usable, continuous memory block.

Figure 8 illustrates fragmentation and the problem it presents. At first we have full utilization of memory. Afterwards we start freeing memory from separate locations, resulting in 2 non-adjacent free memory units. The user sees the amount of free memory is 2 units, but when trying to allocate 2 units the operation fails, as no continuous free memory area of this size is available.

An alternative to the LUT based approach is to pre-allocate the heap into different sized memory blocks, or *slabs* and insert them into a linked list [13]. The maximum allocable amount of memory by a single request is the largest slab size, and vice versa for small allocations. Thus, handling arbitrarily sized allocations are its weak points. Due to storing memory slabs in a list, slab allocation is deterministic timing-wise and does not suffer from fragmentation [7].

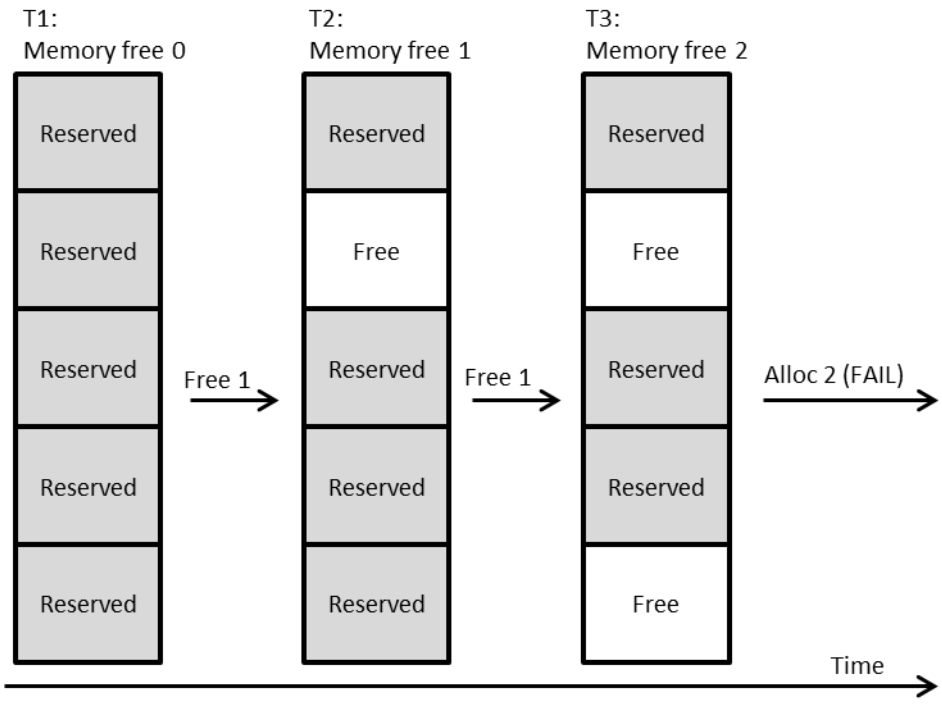

Figure 8.Memory fragmentation and its problem illustrated

2.2.5 Interrupts

A microcontroller unit (MCU) often implements a way to *interrupt* the CPU from running the current program sequence. When an interrupt fires the CPU is *interrupted* and the context is automatically switched from *thread context* to *interrupt context*. Interrupts are designed for hardware to inform the CPU that something has happened that needs attention. An interrupt can be triggered by I/O events or due to a software error, for example illegal access by user applications or an illegal condition.

Interrupts are assigned to *handlers* which are programs designed to service requests from the interrupting hardware [5]. Modern MCUs also implement support for different levels of interrupt priority, by which a higher priority level interrupt can preempt and postpone lower priority interrupts [11].

As an RTOS claims ownership of the interrupting hardware, it must implement the default interrupt handlers. A way for users to *inject* user interrupt handlers is provided. User interrupts are monitored and can be, for example, disabled if they misbehave. This ownership and control is necessary due to RTOS timing requirements. RTOSs also determine interrupt priorities if the hardware supports this.

3. RELATED REAL-TIME OPERATING SYSTEMS

This chapter presents a few popular RTOSs for embedded systems. A short overview for each implementation and its main design principles are given.

3.1 Contiki

Contiki is an open-source, multi-platform, event-driven operating system with multi-threading support, distributed under a custom BSD license. Contiki is energy efficient and lightweight in memory consumption. It can load and unload applications and operating system services dynamically from libraries on reference basis [13].

Contiki introduces protothreads, which is preemptive like cooperative multithreading extension on top of the event-driven kernel. A process is implemented as a protothread, and it is started whenever a process receives an event. Protothreads address the problem of implementing complex state machines on applications to provide concurrency [14].

Contiki tries to provide the best of two worlds by combining the light-weightiness of event-driven kernels, with the scalability, reliability and convenience of preemptive kernels. However, the implementation of protothreads still complicates application design, as applications must define yield points for the protothread library [14].

3.2 TinyOS

TinyOS is an open-source, BSD licensed, event-driven RTOS. It is based on static components that can include tasks, hardware abstractions and algorithms, to name a few. These components are statically mapped and connected with statically linked interfaces. Thus, TinyOS does not support dynamic loading, unloading or communication of processes. Running programs is implemented as a complex network of small event handlers, divided into two segments; a *bottom-half*, or interrupt procedure, and an *upper-half*, or deferred procedure [15, 10].

These segments can be further defined as:

- **Synchronous:** Run in thread-context and these are the upper-half of an event handler, or contain tasks / application code.
- **Asynchronous:** These are run in interrupt-context and these are the bottom-half of an event handler [16].

There is a strict priority relation between the segments; synchronous code can never preempt asynchronous code, but asynchronous code can always preempt synchronous code.

TinyOS and any applications for it are implemented in a dedicated language called nesC. The resulting nesC code is compiled to ANSI-C and furthermore to binary code for the target platform. The nesC compiler optimizes unnecessary parts from the C-code, thus TinyOS is lightweight when it comes to memory consumption. However, as the main drawback application developers must learn the nesC language, that might increase application development times and complexity [15].

As a preemptive multi-threading extension to TinyOS, TOSTThreads was introduced [10]. TOSTThreads extend the TinyOS model to provide a separate context for user applications. User applications are run in a lower priority context, while the TinyOS kernel runs in the highest context. TOSTThreads adds a bit of overhead as a wrapper between applications and the kernel, but application design can be done in ANSI-C which removes the tedious necessity of learning a new programming language [10].

3.3 FreeRTOS

FreeRTOS is an open-source, GPL licenced operating system free to use for commercial applications. The operating system is ported to numerous platforms and in addition to in-house development; partners assisting with development include ST-Microelectronics, NXP, Atmel and Silicon Laboratories to name a few [17].

The operating system is configurable per user need; for instance it supports preemptive, cooperative scheduling or hybrid scheduling, a tick-less mode for low power, strict priority operation and a traditional round-robin scheduling with time slicing to name a few [17]. The operating system also supports non-virtualizing memory management on platforms with hardware support [17].

FreeRTOSs primary design goals are [17]:

- **Usability:** The kernel is implemented in ANSI-C code with a few simple primitives implemented in assembly and the source codes are readily available.
- **Small footprint:** In addition to the highly modular and configurable design, the kernel itself consists of only 3 source modules.
- **Robustness:** Comes from the simplicity of the design.

FreeRTOS also provides a software timer architecture that utilizes the system tick timer for timing. Timers can be created dynamically and time expiration is indicated with callbacks from thread context, ensuring very deterministic operation and very high interrupt service availability as the Interrupt Service Routines (ISR) are kept short [17].

3.4 Micro-Controller Operating Systems (µC/OS-II/-III)

µC/OS-II is a commercial RTOS for embedded microcontrollers. It is portable, supports preemptive scheduling and is provided as ANSI-C source code for licenced customers. The operating system is developed and maintained by Micrium and supports multiple platforms. µC/OS-II scheduling is priority based with a maximum number of tasks each with a unique priority acting as task identifier. Tasks are scheduled whenever a higher priority task decides to yield, with no fairness implemented. µC/OS-II is statically configurable; with similar services to FreeRTOS e.g. software timers [18].

A newer version, µC/OS-III expands the older implementation by offering a ticked mode with round-robin scheduling, unlimited tasks, tasks with the same priority and dynamic configuration. Some of the advanced features include posting messages and signals to tasks without a dedicated message queue or signalling structure, with the option of not invoking the scheduler after posting.

µC/OS-II/-III is also MISRA-C:1998 (Automotive) and DO178B (Avionics) compliant (with a few exceptions). It is also approved for use in medical and nuclear systems.

3.5 Comparison

Table 1 gives an overview of the features and differences of the presented implementations in Chapter 3. N/A means that the information is either not available, or does not apply, for example with event-driven schedulers the concept of tick-less is meaningless. Hybrid scheduling in this context means that the operating system supports either preemptive-, or event-driven scheduling or a combination of both. Static configuration means build-time configuration and dynamic means loading and unloading of modules run-time.

Table 1.Comparison of popular embedded RTOS features

Component	Contiki	TinyOS	FreeRTOS	µC/OS-II	µC/OS-III
Configuration	Dynamic	Static	Static	Static	Dynamic
Scheduler	Event-driven	Event-driven	Hybrid	Preemptive	Preemptive
Tick-less	N/A	N/A	Yes	N/A	Yes
MMU	No	No	Yes	Yes	Yes
License	Custom BSD	BSD	GPL*	Commercial	Commercial
Extensions	Preemptive, Hybrid scheduling	Preemptive scheduling	N/A	N/A	N/A

* Benchmarking forbidden, proprietary sources do not need to be published

3.6 Conclusions

Each of the presented RTOSs is a potential alternative to WPC-OS. However, intrinsic drawbacks in each design motivate the development of a customized RTOS for WPC. Contiki and TinyOS increase software development complexity by introducing an unorthodox scheduling method (protothreads, TOSTThreads) or a new programming language (nesC). µC/OS is commercially licensed and would increase product cost. The FreeRTOS software timers [17] utilize the relatively slow ARM System Timer (SysTick) [30] with a minimum configurable period of only 10 milliseconds [30], which is too long for WPC. The SysTick timer also requires the system high frequency clock to run, and this would increase energy consumption when the system is sleeping.

Modification of the operating system kernels to better suit WPC needs was not considered due to unforeseeable side effects. As an example, modifying the FreeRTOS software timer architecture could produce side effects for the operating system operation when timings get tighter.

4. MOTIVATION AND REQUIREMENTS FOR WPC-OS

This chapter presents a detailed look into WPC and what requirements WPC sets on WPC-OS. The timing schedule followed by WPC is explained and the multitasking aspect is also visited. Some operating system types are ruled out and different feasible options are elaborated upon. The impact of hardware restrictions is also mentioned and the implications to operating system design from the point of view of synchronized connectivity.

4.1 Wirepas Connectivity

Wirepas Connectivity implements a low-duty cycle connectivity protocol specifically designed for battery operated devices and wireless connectivity between the devices. This low duty-cycle operation mode and wireless medium access is called *Contention-Free Medium Access Control* (CF-MAC).

The access is divided into discrete time slots, or *superframes*, repeated on a predetermined time interval, or *access cycle*. The superframes are repeated simultaneously on several frequencies by different devices. This type of time/frequency division is called *Frequency-Time Division, Multiple Access* (FTDMA). Superframe repetition and TDMA slots on a single frequency are illustrated by Figure 9.

The devices in the network are called *nodes* and these nodes have two different roles: *cluster heads*, and *cluster members.* Each cluster head maintains a superframe at a given FTDMA slot. A cluster head owns the FTDMA slot and cluster members participate in the network by accessing clusters during their superframes.

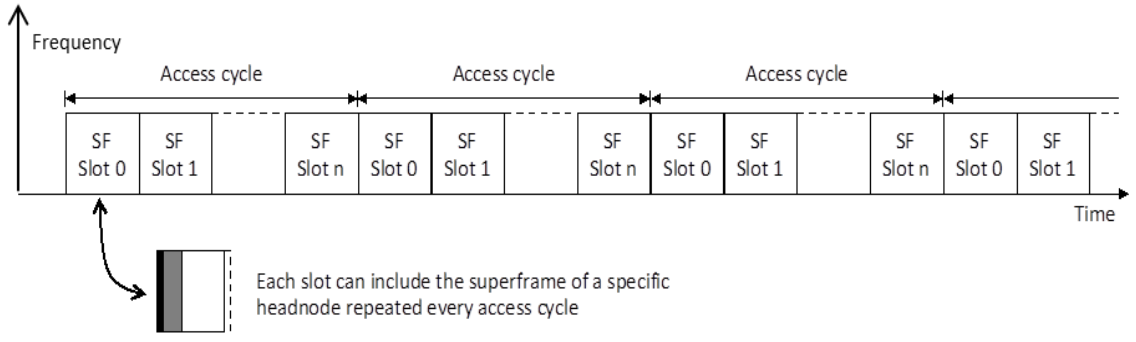

Figure 9.Superframes repeated every access cycle

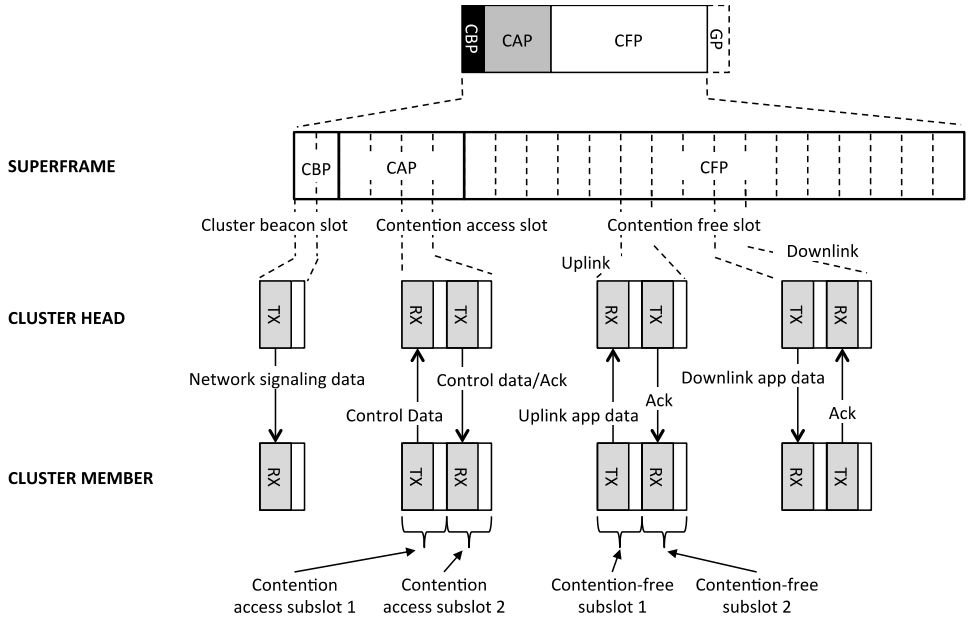

Figure 10.Superframe structure

The superframe consists of four components, or periods, illustrated by Figure 10:

- **Signaling period (CBP):** This begins the superframe, and is used to propagate common information about the cluster head to listeners. This signaling happens at a precise moment, repeated every access cycle.
- **Contention Access Period (CAP):** This is used for control messaging from cluster members to the cluster head. Contention means the time slots are not owned by any particular cluster member, so collisions can happen.
- **Contention Free Period (CFP):** This is used for data exchanges between the cluster head and cluster members. Data can flow in both uplink and downlink direction in CFP slots.
- **Idle / guard-period (GP):** This period is optional and used to ensure compliance with regulation.

All information transactions happen inside pre-determined timeslots and the beginning of a slot, or slot *boundary*, is the timing critical moment. Within a slot one node transmits (TX) information while another node (or many) is receiving (RX) and both operations are synchronized to this slot boundary. This results in low duty cycle operation both TX and RX wise, increasing energy efficiency. Missing the slot boundary time due to timing inaccuracy by either party will result in a failed transaction.

Figure 11 presents what happens at a slot boundary from both perspectives. Timing inaccuracy is compensated on the receiver's end by pre- and post-boundary margins. The receiver is turned on before the slot boundary and is kept on after. Better timing accuracy means tighter margins can be used, resulting in better energy efficiency.

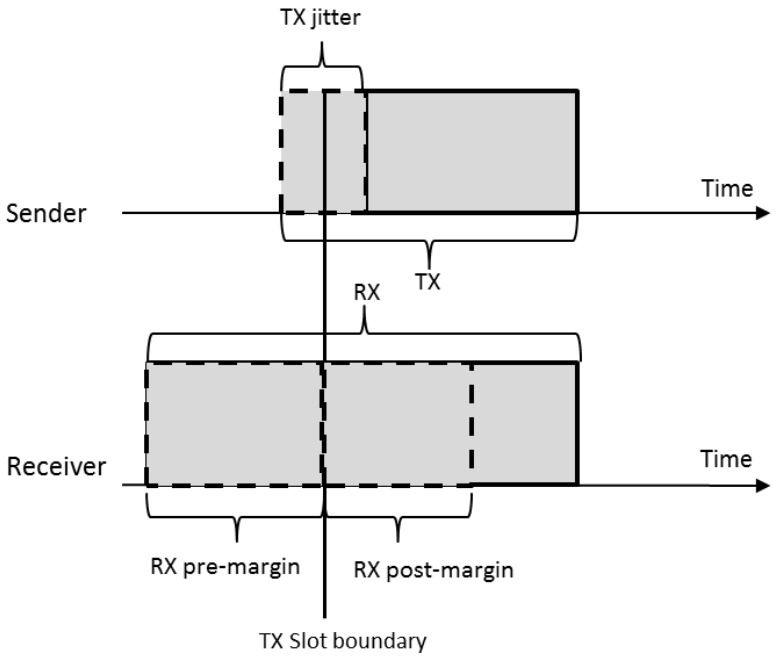

Figure 11.Slot boundary with TX and RX margins

CAP and CFP slots are divided into two subslots. The first subslot is used for transmitting information, and the second subslot is used for acknowledgement. A transmission is repeated until an acknowledgement is received, thus no information is lost. This means that if a slot boundary time is missed and the transmission fails, it is attempted later.

This retry mechanism also applies to CBP slots; if the cluster heads CBP is missed, the cluster member will attempt again when the CBP period is repeated in the next superframe. Therefore, consequences of missing a timed network event are not fatal and thus WPC can be qualified as a soft real-time-system. Re-attempting however wastes energy and increases latencies, so missing deadlines consecutively is not acceptable.

Another relevant part of the protocol is the neighbor discovery protocol and discovery period. During a discovery period the node is searching, or *scanning* for other devices and attempting to join, or *associate* with cluster heads. When other devices are found, the device attempts to *synchronize* with them by participating in a cluster heads signaling- and contention access periods. The synchronization information is distributed by cluster heads periodically and the timings are calculated locally by cluster members from this information.

The discovery period implies long time periods of processing for CF-MAC, especially when the device is scanning. For the synchronization to be accurate, accurate time keeping and timed events are also critical for CF-MAC operation.

4.2 General requirements for the kernel

The WPC stack follows in the footsteps of Wireless Sensor Networks (WSN) and thus many of the requirements are the same. The requirements can be generalized as follows in priority order [9]:

Real-time operation:

WPC is a time sensitive protocol, meaning all operations are strictly timed within the network. WPC deals with time in microsecond accuracy, and all timing critical operations follow this same granularity. With the node having to optionally serve several timing critical interfaces, real-time operation of at least the radio part must be ensured.

Energy management:

As WPC is designed for battery operated devices, the operating system should handle energy management to achieve as low power consumption as possible. As synchronous digital logic consumes energy mainly when it changes states, energy efficiency can be achieved by shutting down unused peripherals and entering deep sleep if no peripheral requires the system master clock.

Small program and data memory footprint:

Due to resource constraints in embedded systems, the operating system footprint should be as small as possible. As example hardware the Nordic Semiconductors nRF51822 has 256 kB of program memory and 16 kB of data memory. WPC itself consumes 82 kB of program memory and 7 kB of statically allocated data memory, reducing the available memory for WPC-OS. The scheduling method also impacts data memory usage. Preemptive scheduling uses a lot of data memory as each task is run in its own unique sandbox and thus requires its own stack. If the amount of tasks is high (in the order of tens to hundreds) a preemptive scheduler might not be feasible, as the amount of memory required is increased linearly by each task. As an example, if the system requires 10 tasks with 256 bytes of stack memory, a total of 2.56 kB of data memory is already used by the operating system.

Multitasking capability:

A WPC network is concurrent by design. Data is exchanged concurrently between nodes and must happen in a timely manner. The ability to run multiple custom applications and concurrent servicing of other wired- or wireless protocols is necessary. Thus, priority based multitask scheduling, IPC messaging and timing mechanisms are required.

Memory management:

Smart utilization of the limited data memory is required to handle the challenges of unpredictable and sporadic data exchanges between nodes and the application. The operating system designer cannot know the nature of data memory usage or data transmission patterns of an application in advance, so dynamic memory management is required.

Peripheral access:

As the applications used with WPC might have the same hardware requirements with WPC-OS, an unambiguous access to shared hardware is required. A shared timer is the most common requirement, and requires centralized control. To prevent the applications from interfering with operation of the stack, the operating system should take ownership of shared resources and handle controlling them for safe use.

Modularity and hardware abstraction:

To better utilize constrained resources on the platform, the OS should be configurable. Task amounts, events, memory maps etc. should be configurable per application need. As WPC is defined as platform independent software, hardware abstraction is needed for portability and ease of access to hardware. The Hardware Abstraction Layer (HAL) should be well defined for fast implementation of new platforms.

Easy application design:

The pressure to keep time to market short on IoT solutions is high [3], so time spent on application development should be kept at a minimum. A well-documented, easy to use interface decreases application design time.

Most existing RTOS kernels are static systems, meaning that all resources are pre-allocated during build time, and run-time customization is not possible. This static approach offers many benefits though; reserving and freeing resources is deterministic timing-wise and resource allocation does not waste CPU time or energy. Static systems can also be more reliable, as the system resources are known in advance. This however requires that the operating system is configured correctly per the application needs.

Dynamic reservation of resources offers easier application development, but as the resources are implicitly limited by hardware limitations, problems can still occur if the system resources are overloaded. However, if a dynamic system is used correctly, it can be efficient in very simple applications that use a very little amount of operating system resources.

4.3 Timing requirements for the kernel

Timing requirements are the main motivation for WPC-OS. The platform must be capable of running WPC alongside serving several asynchronous interfaces:

- **User interfaces:** Buttons, Light Emitting Diodes (LED), displays, speakers
- **Serial interfaces:** Bi-directional (Full-Duplex) Universal Asynchronous Receiver / Transmitter (UART), Serial Peripheral Interface (SPI).
- **Network interfaces:** Such as Ethernet
- **Sensor measurements:** Real-time Analog To Digital (ADC) conversion

These interfaces require timely servicing from the system. Otherwise, user inputs, data from serial interfaces or Ethernet packets can be missed.

With WPC, all communication is tightly synchronized and transmissions are expected only at slot boundaries. This implies the need for accurate timings and deadlines for the CF-MAC. To keep time management deterministic in a platform performing operations on multiple interfaces, using interrupts for event handling is critical. All protocol timings must be interrupt-driven and performed in the highest interrupt context. Radio packet reception should be indicated by the radio via interrupt for accurate packet timestamps. Enabling the transmitter and receiver should be done via timed events in interrupt context to prevent any other asynchronous event from interfering with the timing of the radio operation.

For energy efficient operation, it must be possible for the system to sleep in the lowest possible sleep state with timed wake-up support. A Real-Time Clock (RTC) is capable of running in low power modes and waking up the system from these modes. RTC timing is done with an external 32.768 kHz crystal oscillator (XTAL). This results in a period of 30.5 microseconds (us) per XTAL tick. For real-time events the timing error must be below 1 XTAL tick, and for generic system events an accuracy of ±3 XTAL ticks is required. These figures come from the timing requirements of the CF-MAC, which is tightly optimized timing-wise for energy efficiency.

4.4 Scheduling options for WPC

Viable scheduling options for WPC-OS are *event-driven* scheduling and *preemptive* scheduling. A preemptive scheduler is deterministic for the timing requirements of WPC, but as missing deadlines is not fatal, a non-preemptive scheduler is also viable.

As a standalone solution WPC could also run as a simple *scheduling loop*, but this will not offer sufficient concurrency if applications using external interfaces are introduced, so this approach is rejected.

4.5 Preemptive scheduling for WPC

Preemptive scheduling offers a seemingly seamless execution of multiple tasks on a shared hardware. This means the following requirements of WPC are the strong points of a preemptive scheduler [9]:

- **Easy application design:** Application designers do not have to worry about the timing of the application and interfering with the stack's operation.
- **Real-time:** Priority based preemptive scheduling ensures a fast context switch to the stack from the application whenever the stack requires CPU time.
- **Reliability:** As a lower priority task can be suspended whenever, the stack should never miss deadlines due to misbehaving applications.

However, there are downsides to the preemptive scheduler for WPC:

- **Energy efficiency:** Every context switch uses energy. Applications run with WPC are usually very simple and don't necessarily require a fast context switch.
- **Data memory footprint:** As with a preemptive scheduler every task requires its own software stack, the data memory consumption of the kernel can be high compared to the overall amount of available memory.

4.6 Event-driven (non-preemptive) scheduling for WPC

Due to less context switches, non-preemptive scheduling is better with energy management. A non-preemptive scheduler needs only one software stack thus it has a smaller memory footprint. As a downside, application design gets harder as all tasks are run to completion. This means the application must handle long processing needs itself and must yield execution in time for the WPC stack operations to run on time.

The application designer must measure the time the application uses in a worst case, or implement a complex state machine to handle long processing needs. However, measuring the worst case processing time is not always feasible due to, for example, different platforms with different processing power. Thus, interfering with the protocol timings is not always avoidable [9].

4.7 Resource constraints for WPC

As WPC is designed to run on simple embedded systems, the resource constraint aspect dominates the kernel design. Most MCUs for embedded systems implement a Reduced Instruction Set Computer (RISC) and most modern MCUs implement an Advanced RISC Machine (ARM) [29] processing unit. WPC is designed exclusively for ARM, which means the main differences and strong points between platforms are the program and data memory amounts, available peripherals and especially energy consumption.

These platforms are usually complete System-on-Chip (SoC) solutions including synchronous and asynchronous serial ports, Analog to Digital Converters (ADC), General Purpose Input / Output (GPIO) blocks, Advanced Encryption Standard (AES) accelerators for security, Co-processors or Peripheral Reflex Systems (PRS) for reading sensors without waking up the CPU, and, optionally, a wireless radio. The currently supported WPC platforms and their features are listed in Table 2 [21, 34-39].

The memory amount figures place constraints on WPC-OS design. It is not feasible to use all available memory for the kernel implementation. The design must also take into account the memory needs of multiple user applications and WPC.

Table 2.List of supported WPC platforms

MCU	Maximum program memory (kB)	Maximum data memory (kB)	Radio	AES	Co-processor	Maximum CPU frequency (MHz)
SiLabs EZR32LG	256	32	Yes	Yes	Yes	48
SiLabs EFR32	256	32	Yes	Yes	Yes	40
STM32F and L-series	1024	196	No	Yes	No	168
Nordic Semi nRF51822	256	32	Yes	Yes	Yes	32
Nordic Semi nRF52832	512	64	Yes	Yes	Yes	64
TI CC2650	128	20	Yes	Yes	Yes	48

Battery operation means that energy consumption must be kept at a minimum by using the absolute minimum amount of resources on the MCU. As digital logic consumes power when changing states [2] (i.e. on a clock edge), selecting the correct running frequency for the CPU is critical, and the tradeoff between completing tasks to enter sleep faster and consuming more power when running has to be evaluated. Automatic clock gating and shutting down the CPU are essential, especially if the system is running at its maximum frequency. To preserve energy, the system should also enter the deepest possible sleep state depending on present conditions, if it has nothing useful to do.

Modern ARM MCUs also include rudimentary memory protection. The Cortex-M series implements a Memory Protection Unit (MPU) that separates kernel space from user space [11]. The more powerful Cortex-A series cores implement a MMU for full virtualization needed by traditional GPOSs [28]. This provides the option of implementing memory protection in WPC-OS, but it is not a WPC requirement.

A summary of the requirements is described in Table 3. Only requirements relevant for WPC operation are covered.

Table 3.Summary of WPC-OS requirements

Requirement	Details	Target
Real-time event accuracy	The real-time event service must be accurate and deterministic. No other interrupt sources must interfere with the timings. The protocol benefits from accurate timings by allowing tighter margins with less missed transmissions, resulting in better energy efficiency.	Accuracy of below one XTAL tick (30.5 us) under all conditions
Energy-efficiency	For battery operation, the operating system itself should be as lightweight as possible.	Minimize cumulative CPU active times
Tickless	The system must operate on pure event-driven basis, with no unnecessary context switches or wake-ups.	Minimize unnecessary wake-ups
Optimized memory footprint	The limited program and data memory resources on supported WPC platforms dictate the available memory for the OS.	Usages: Program memory < 10 kB Data memory < 1 kB
Automatic energy management	The operating system should offer a coherent way of handling CPU power states and peripheral clocks. The optimal CPU power state must be determined dynamically per each sleep period. Peripheral functionality must not be affected by CPU sleep states.	Implement idle task and a centralized energy management module
Ability to run multiple tasks	At least the mandatory protocol stack tasks; MAC, routing, management, maintenance and an application interface support task are required. On top of this, one user application, and four stack support applications are required.	Minimum amount of tasks: 10 tasks + idle task

5. WPC-OS DESIGN

This chapter presents the WPC-OS design. A look at the RTOS services and architecture is given. Finally an overview of all relevant modules, including the critical HAL components is explained.

5.1 Design and architecture

The main design idea and motivation for WPC-OS was to create a simple solution to concurrency problems presented by platforms with multiple asynchronous interfaces. As the protocol stack requires timing accuracy in the microsecond domain, lengthy operations run in interrupt context must be preemptible by the high priority timing services, or by radio events. It should also be possible to split these operations into two parts, upper-half and bottom-half, for better utilization of CPU for the really timing critical parts.

WPC-OS architecture is presented in Figure 12. WPC-OS is a *static* implementation, meaning the operating system cannot be configured at run-time. System resources, excluding dynamic memory, must be reserved before the scheduler is started, meaning dynamic loading and unloading of tasks is not supported. This is supposed to bring predictability to the system operation, as the system stays in a static formation after started. Hardware resource consumption can also be determined precisely at build-time, and if the configuration is using up more resources than possible, the system designer knows immediately that something is wrong.

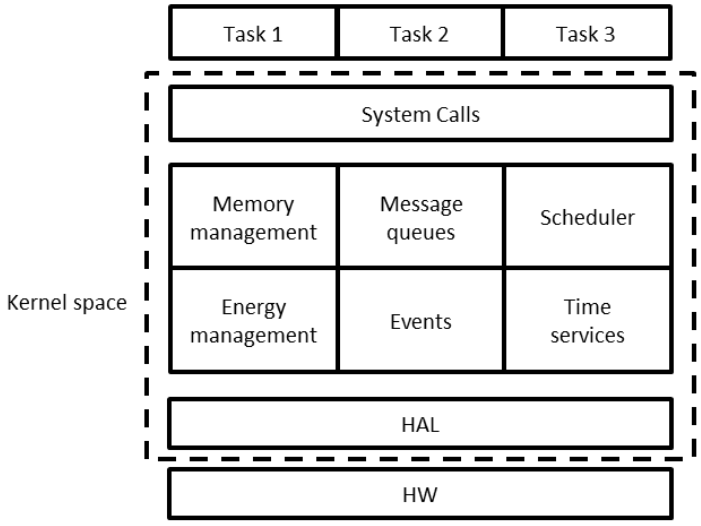

Figure 12.WPC-OS architecture

The WPC-OS kernel provides two fundamental timing services; real-time events usable only from a single task with sufficient priority, and a software timer interface that is usable from any task. Services are supplemented with basic dynamic memory management, message queues for processes, energy management for power saving states and control, and events. All this is built on top of a cooperative, non-preemptive task scheduler.

5.2 Tasks

Tasks in WPC-OS are typical functions; they have a beginning and end. The tasks tell the scheduler the interval at which they require CPU time, and the worst case duration of the task. This worst case duration must be measured by application designers. Measuring the worst case duration is difficult, if the application does iterative work or if the application is intended to work on platforms with different processing power. Thus, the measurement may include some error and this must be taken into account by adding margin when providing the task duration for the WPC-OS kernel. If the provided task duration figure is smaller than the actual duration, the timely scheduling of other tasks will no longer work.

Each task is assigned a PID and given a symbolic name for debugging convenience. This PID is used for claiming ownership of resources by the task, and by the kernel to check privileges to run operating system services.

Each task also has a static priority, with two unique priority levels:

- **Real-time:** this is reserved for one task only. If the real-time task has something to do, no other tasks are considered. Additionally, only this task has privileges to access the real-time services offered by the operating system.
- **Idle-task:** This is run whenever the system has nothing to do, and there is sufficient time to deal with task execution turn-around delay and wake-up event handling.

Each task can be woken asynchronously by signaling the task from any synchronization structure. The kernel maintains these signals for scheduling purposes.

Tasks also have a dynamic priority component, called *timetable* priority. This priority level indicates to the operating system that the task has periodic work to do, and should be awoken at a certain interval. Tasks themselves can attempt to change this dynamic priority via service call. Tasks without a timetable are asynchronous tasks that will never run unless they receive a signal from an event handler.

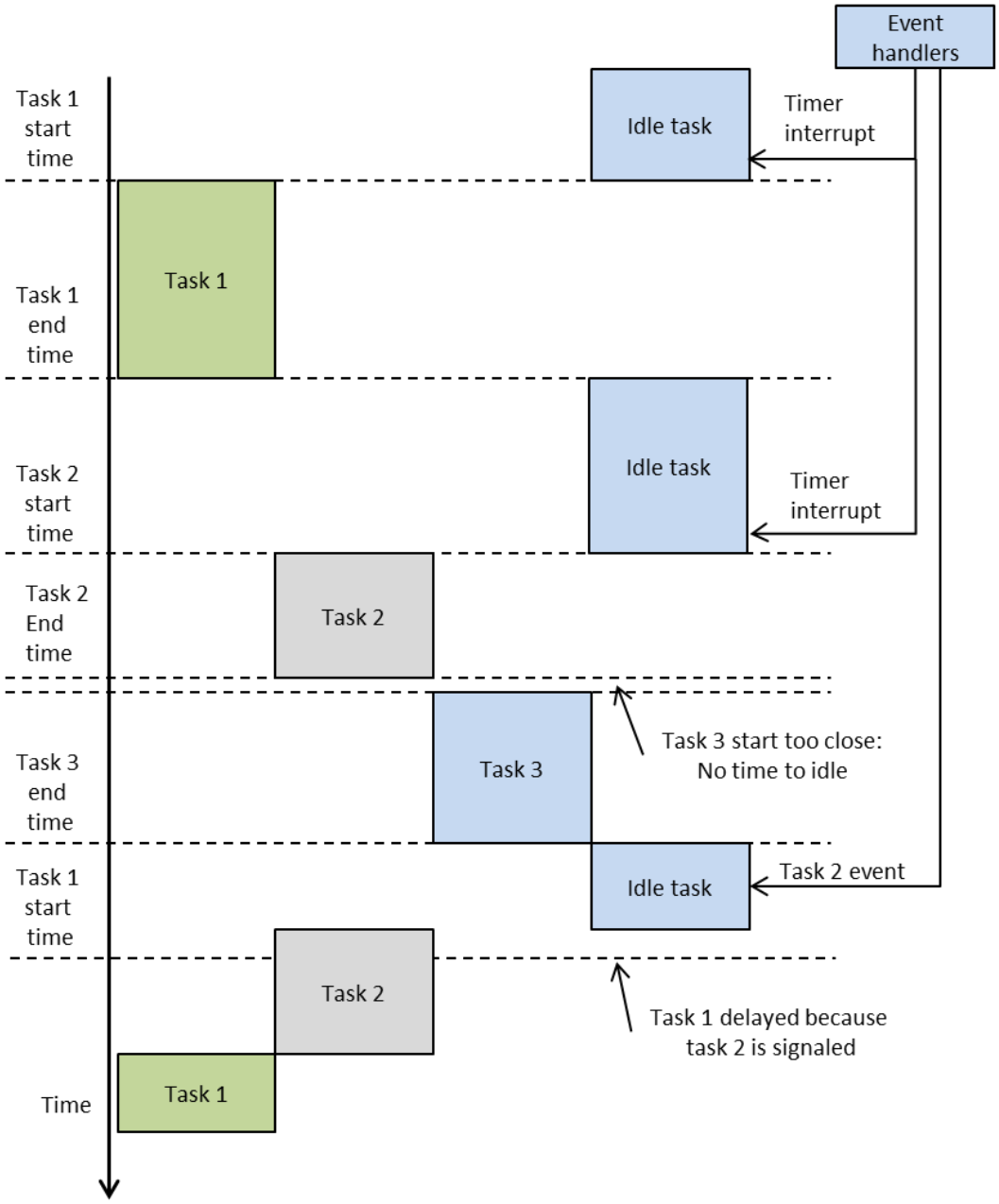

Figure 13.Task scheduling example with priority elevation by asynchronous event

5.3 Scheduler

The scheduler is a priority based, non-preemptive control loop with asynchronous task scheduling support. Multiple tasks with equal priority are also supported. Task scheduling is not designed to be fair, with the exception of tasks with equal priority. With equal priority tasks the scheduler targets to fairness. Functionality of the scheduler and running tasks at a given time instance is described in Figure 13. It calls task entry points in a timed manner, with added event reactivity.

Task scheduling follows the *shortest task first* principle [7], meaning if there is time to execute short tasks before a higher priority task, the execution order is dynamically changed to perform the short task first. Tasks tell the scheduler when they want to be executed, and the longest possible execution time of the operation.

With this information the scheduler compares start times and durations of all tasks and finds the best suiting task to be executed at any given time instant. Task priority is also elevated dynamically by asynchronous events, and a task signaled by such events always gets priority over pure timetabled tasks. A pure timetabled task in this context means a task that does not own any event handlers, and just follows a periodic timetable to perform work.

The idle task is also considered a part of task scheduling; if the system has nothing useful to do, all wait states are performed by the idle task. In the idle task, the system enters the deepest possible sleep state to preserve energy, awoken by either a timer interrupt, or any asynchronous task event. The idle task is scheduled if there is enough time for the idle task to run; that is setting up a wake-up event, exiting sleep in a safe manner and the turnaround delay of a new scheduling loop.

5.4 Services

This section describes the common RTOS services provided by WPC-OS: events, timers, message queues, hardware access via drivers and dynamic memory management.

5.4.1 Events

WPC-OS provides a simple and coherent event service for task synchronization. Due to the static nature of the OS, the maximum amount of events must be decided at build-time but the events can be reserved and freed at run-time. When an event is reserved, a task will claim ownership of the event. This implicitly means that only one task can be waiting for an event to occur, so multi-task synchronization via this mechanism is not supported.

The event mechanism is divided into two parts; waiting and signaling. Waiting is done in thread context and the signaling occurs from interrupts or kernel events such as message queue push events. Waiting for an event blocks execution of a task and it enters a low power mode until the event is signaled, or the wait state is timed out. Two timeout values for polling and infinite wait are provided. Polling just returns the event status. Infinite wait enters a low power mode and stays there until the waited event is signaled.

Another sub-category of an event is tied to the Task Control Block (TCB) itself. A task can be signaled by kernel events to notify the scheduler that the task has something to do. Usually kernel events are set by messages sent to the task.

5.4.2 Timers

The operating system provides two timing mechanisms; timeouts and real-time events. The timeout mechanism is a multi-user interface for any type of timings, with limited accuracy. Real-time events are a single user interface with ultimate predictability and timing accuracy.

Timeouts are provided by a special operating system service call API called *timeouts*. Multiple active timeouts are supported, with the same system-time instance as the event time. These timeouts are not directly user-accessible, and form the backbone of all timed operations within the system. For example, the scheduler and the idle task utilize this service to time the beginning of tasks correctly. The guaranteed maximum bias is +3 XTAL beats (91.5 us).

The real-time timer service is reserved for the real-time task only. This is an extremely light-weight interface to the real-time timer. The interface provides a mechanism for setting a timed event with one microsecond (us) accuracy, and a dedicated wait function to enter a low power mode that waits specifically for this real-time event to occur. The event can also be tied with a callback to user space, to perform the timed operation in the highest possible CPU context so that no other task can interfere with the operation.

5.4.3 Message queues

Message queues provide messaging and synchronization between tasks. The queues bind the message queue itself to a task and the scheduler in an abstract manner. When a message queue is created, ownership is also claimed by a task. When a message is received, it is stored in the queue structure and the task and the scheduler are notified. This provides the possibility of scheduling the receiving task as fast as possible to react to the message. The addressing of messages is done by the unique PIDs which must match the owner of the queue.

5.4.4 Drivers

The kernel itself is implemented as a microkernel, which means it does not claim ownership of hardware it does not require itself. The kernel itself does not implement any drivers, and all hardware is accessed via the kernel HAL. The kernel HAL only includes timer drivers, CPU power states and deep sleep control and interrupt handling. This segregation makes porting of heterogeneous platforms easier for WPC-OS.

The kernel HAL drivers do not need anything from the kernel, and thus could be utilized bare-metal. However, if WPC-OS is used, the necessary driver resources are reserved by the kernel and can only be accessed via WPC-OS services.

5.4.5 Memory Management

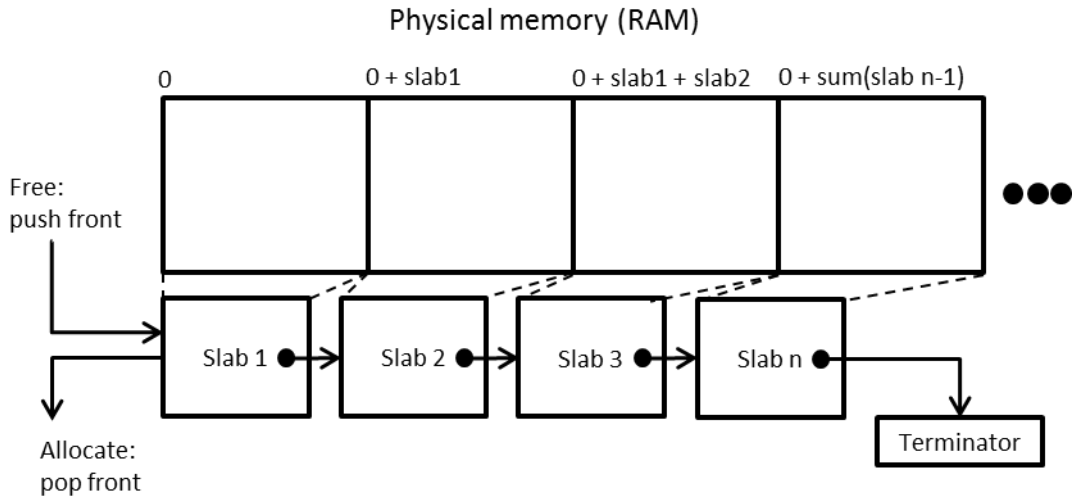

Figure 14.Dynamic memory management by slab allocation

Memory management provides dynamic memory allocation and freeing services to tasks. Dynamic memory is implemented as fixed size slabs, inserted into a linked list structure during the operating system initialization. The operating system supports three different slab sizes each with configurable amounts and sizes that can be tweaked by user demand. The memory management also provides memory usage diagnostics in form of usage counters, allocation utilization and reservation success rates for system designers.

This fixed size slab approach provides a deterministic operation time of O(1) for both allocating and freeing blocks. Allocating memory is a pop-front operation, while freeing is push-front, both just changing the linkage of the first element. In addition to being deterministic timing-wise, this approach does not suffer from memory fragmentation. Figure 14 illustrates how the dynamic memory is constructed for one slab size configuration, and how allocation and freeing only affects the linkage of the first element.

The weakness of this fixed size slab approach is handling dynamic memory needs in a system requiring arbitrarily sized memory blocks. The smallest amount of memory allocable in a single request is always the smallest slab size, and vice-versa the largest block is limited to the largest slab. Regardless, the user can demand for any sized blocks, and in this case the best fitting block is allocated.

Another weakness is the non-optimal utilization of memory for small allocations. This implies responsibility on the system designer to select the slab sizes in an optimal manner. For example, WPC is a protocol with a maximum sized protocol data unit so selecting a slab size that corresponds well with this maximum boundary is an option. IPC messages might require another type of container, so offering a few slabs with this size is a good idea.

To keep the memory management simple, fast and predictable, no memory protection is implemented. All WPC supported platforms implement a MPU, but implementing protection would increase memory usage and processing overheads with little benefit.

5.5 Interrupts

Interrupt handling in ARM MCUs is provided by its Nested Vectored Interrupt Controller (NVIC) [43]. It supports up to 240 external interrupt sources and 256 different priority levels [43]. NVIC also supports preempting lower priority interrupts by higher priority interrupts. This mechanism is utilized in WPC-OS design by prioritizing WPC related exceptions, such as system timer events.

Interrupt vectors are provided for NVIC as a vector table. This contains addresses for the interrupt handlers. When an exception occurs, the NVIC automatically saves the current CPU state, and the program execution jumps to the address provided in the vector table.

The WPC-OS kernel does not take ownership of any interrupt vectors other than the system timer interrupts. However, the kernel sets up and maintains the NVIC priorities in such a manner that the kernel timing services are always the highest priority interrupt sources in the system. The global interrupt enable flag is also accessed via the kernel, with a reference counter keeping track of atomic sections.

6. WPC-OS IMPLEMENTATION

This chapter presents the implementation details of WPC-OS. Only the relevant parts will be presented. The timer HAL is considered a critical part for the RTOS kernel and thus is presented as well.

6.1 Programming language and compiler

The kernel, kernel HAL and all WPC-OS services are implemented in the C-language, ANSI C99 [19] dialect with some GNU extensions. This is compiled with the GNU C (GCC) [20] compiler for ARM target, while the platform used for this thesis is the Nordic Semiconductor nRF52832 SoC [21], with a Cortex-M4 ARM CPU [42].

6.2 Kernel

The kernel manages tasks, their statuses, durations and priorities. The kernel initializes all operating system modules, offers a way to create tasks and handles the current task. The operating system services gain access to TCB information via the kernel.

6.2.1 Tasks

Program listing 1 introduces the TCB for the kernel. The TCB contains task priority, status, PID, the execution function and timing information for the scheduler. The timing information is in microseconds.

The task start time is maintained by the scheduler. This is the next execution start time for the task. The interval value is provided when the task is created, and the scheduler attempts to provide this timing for the application. Providing the interval information is not necessary, if the task does not set the timetable priority bit. Clearing this bit means the task is purely asynchronous and thus does not require timed execution.

The tasks worst case duration must be provided regardless of task type, so that the scheduler knows when it is safe to execute the task. Measuring this can be done in several ways. One way is to use the system timer to obtain timestamps at task entry and exit, and subtract. Another way is to indicate task entry and exit with GPIO pins and measure the duration with, for example, an oscilloscope. The measurement might contain some error and thus some margin must be added to the measured duration. If the provided maximum duration figure is too short, the timed execution of other user tasks might not work. If it is too long, the tasks execution might get skipped unnecessarily.

```
typedef struct
{
    const char *    task_name;        //< Symbolic name of task
    os_prio_e       priority;         //< Priority of task
    volatile union
    {
        struct
        {
            uint8_t skip     : 6;     //< Task skipped counter
            uint8_t signaled : 1;     //< Task signaled flag
            uint8_t timeout  : 1;     //< Task timed out flag
        };
        uint8_t     status;           //< Task status register
    };
    os_thread_id_t  id;               //< PID of task
    uint32_t        (*execute)(void); //< Task thread of execution
    uint32_t        start;            //< Task start time in system time
    uint32_t        interval;         //< Task start interval
    uint32_t        duration;         //< Task worst case duration
} os_task_t;
```

Program 1. *Task Control Block (TCB)*

A thread id, or PID is defined as:

```
typedef uint16_t            os_thread_id_t;                                 (1)
```

In theory, the system supports 65535 tasks. In practice, this amount is limited at build time to the number of tasks required for each target. The kernel maintains a LUT of all processes, and this PID is the index in the LUT where the tasks TCB can be found.

Program listing 2 declares task priorities, and the timetable flag. Priorities are inversed, meaning lower number means higher priority. Consequently, a task with the timetable flag set is always of lower priority to a task that is purely asynchronous. The real-time priority is reserved for one task only; in the case of WPC the CF-MAC reserves this priority level.

```
typedef enum
{
    /** Task has a timetable */
    OS_PRIO_TIMETABLE   = 0b00000001,
    /** Highest task priority */
    OS_PRIO_REAL_TIME   = 0b00000010,
    OS_PRIO_HIGH        = 0b00000100,
    OS_PRIO_NORMAL      = 0b00000110,
    /** Lowest task priority */
    OS_PRIO_LOW         = 0b00001000,
    /** Reserved for OS idle task */
    OS_PRIO_IDLE        = (0xFF & ~(OS_PRIO_TIMETABLE))
} os_prio_e;
```

Program 2. *Task priorities*

6.2.2 Scheduler

The scheduler is started with a call to *Os_schedulerStart( )*. This function never returns and is never re-entered. All tasks, including the idle task are executed from within this context.

The scheduling decision is done after a task has completed, and is done in two parts. The first part tries to find the best fitting task for the current *time slice*. The task start time and duration are used for this evaluation. Timetabled tasks start times are updated automatically, while asynchronous tasks are scheduled as soon as possible. Task priorities affect this decision, and if two tasks with different priority have work to do, the higher priority task is always scheduled first. Tasks with equal priority have a *task skipped* counter, and the task that has been skipped more gets execution priority for fairness.

The second part evaluates whether there is time to run the idle task before the current task execution. If there is time, the scheduled task is replaced by the idle task. Calculating the idle period length is easy:

$$T_{idleperiod} = T_{taskstart} - T_{now} , \tag{2}$$

where $T_{idleperiod}$ is the idle period duration, $T_{taskstart}$ is the current tasks start time and T_{now} is the current system timestamp. Idle period is scheduled if the idle period duration is less than idle period turnaround time. The turnaround time includes the idle task turnaround itself, plus the scheduling of a new task. A new scheduling decision must be made as the idle period can be interrupted by an asynchronous event for a higher priority task.

6.2.3 Events

Program listing 3 declares the event structure for the kernel. For users only a handle is declared, which points to the kernel space event. Events can be allocated and destroyed while the operating system is running. The allocation ties ownership of the event and resets the event signal, returning a handle to the event. Afterwards the event can only be accessed via a system call giving the handle as a parameter.

```
1  typedef struct
2  {
3      os_thread_id_t  thread_id; //< Owner of event
4      volatile bool   signaled;  //< Status of event
5      uint8_t         unused;    //< For EABI alignment
6  } os_event_t;
```

Program 3. *Event declaration*

Events can be used to synchronize tasks to asynchronous events from hardware, and implementing energy saving states. The usual use case is waiting for e.g. a radio packet reception to complete. The task enters a wait state and the radio packet handler interrupt sets the event when a packet is received. Also, due to the synchronous nature of WPC there must be a way to cancel the wait state when data is no longer expected.

Waiting for an event is done via system call:

```
bool Os_waitEvent(os_event_handle_t handle, uint32_t timeout)          (3)
```

Where handle points to the event and timeout is the time to wait for the event to occur in microseconds. The wait state can be of infinite length, which means the system waits forever for the event to occur, or zero-length which means the event status is returned immediately. If a timeout is requested, the kernel reserves a timeout from the kernels timeout timer pool. The functionality of this timeout mechanism is explained later.

When waiting for an event and if the event has not been signaled, the system enters a low power mode where it can only be woken up by interrupt. This interrupt can either be the event itself, some other event or in case a timeout for this wait state is needed, the kernels timeout service ending the wait state. The function re-enters the low power mode if the wake-up was caused by an unrelated event.

The idle task abuses the event mechanism by waiting for a special idle task event while giving the idle task duration as the timeout. The idle task event is signaled by all events, allowing preemption of the idle task if something important happens, while ensuring the idle task is exited by timeout when the idle period is over.

6.3 Timers

The operating system timers are split into two services, one for alarm-type signaling called *timeouts* and one for executing real-time events in the highest priority interrupt context. The implementation is split into two layers; the kernel system call layer and the platform dependent timer HAL. The kernel layer is a light abstraction used only to provide safe concurrent access to the shared hardware and determining permissions when using the real-time timer interface.

6.3.1 Timer HAL

The timer HAL is called *rtimer*, and requires a minimum of two low frequency Real-Time-Clock (RTC) compare match channels and a high frequency general purpose timer (GPTIMER) with one compare match channel. The RTC channels are used for low power timings with 30.5 us (one XTAL tick) accuracy, while the GPTIMER channel is used to achieve timings with microsecond accuracy. The components of rtimer are:

Local system timekeeping with microsecond accuracy

The system time is maintained by the RTC and a high frequency timer in full 32-bit timestamp format, where one tick corresponds to one microsecond. The RTC maintains the actual system time, while the high frequency timer is used only when necessary to achieve the full resolution. The semantics of system time up keeping are entirely platform dependent and will not be elaborated here further.

Timeout / alarm service for low priority, low timing accuracy events

This provides access to one RTC compare match channel. A callback must be registered to the HAL module before usage. This is to ensure maximum performance as the callback does not have to be dynamically set and released for each event. When a timeout is requested it sets a compare match event relative to the current system time. The compare match interrupt invokes the callback notifying the client that this time has passed. As the timeout mechanism utilizes RTC only, the resolution is one RTC tick (30.5 us).

Real-time service for high priority, high timing accuracy events

This mechanism reserves one RTC and one GPTIMER compare match channel to achieve the best possible accuracy while maintaining low power operation. The event time is given in absolute system time format, requiring that the user obtains a system timestamp and calculates the offset beforehand.

When setting a real-time event, the service calculates a dual stage timing mechanism using both RTC and the GPTIMER. A RTC compare match interrupt is set to the event time minus a GPTIMER pre-margin. The RTC interrupt then synchronizes to the system time and sets a GPTIMER compare match event to happen precisely when the user has requested it. The GPTIMER pre-margin is entirely system dependent and has to take into account system wake-up delays and the need to synchronize to RTC time after the wake-up. If the requested time is closer than the GPTIMER margin, the RTC part is skipped.

This service also requires that a callback is set before usage. The user callback is invoked from the GPTIMER interrupt which is always the highest priority interrupt in the system, ensuring absolute predictability and accuracy. This is later on tested and both aspects are proven.

6.3.2 Timeouts

Program listing 4 declares the private and public interfaces of OS timeouts for the kernel and user respectively.

The timeout service handles concurrent access to the system timer hardware so that multiple users can set timeouts, even with the same time. The timeout service provides a parameter structure that is entirely user defined, and these parameters are then passed back to the service caller when the time has passed.

```
1  // Public service call parameter structure
2  typedef struct
3  {
4      os_timeout_cb_f cb;      //< Mandatory user callback for timeout
5      void *          p1;      //< Optional parameter passed to callback
6      uint32_t        p2;      //< Optional parameter passed to callback
7      uint32_t        timeout; //< Timeout time in microseconds
8  } os_timeout_param_t;
9
10 // Private declaration of timeouts
11 typedef struct
12 {
13     os_thread_id_t          owner;     //< Owner of timeout
14     uint8_t                 padd[2];   //< Mandatory EABI padding
15     os_timeout_param_t *    svc_param; //< Parameters for timeout
16     rtimer_timeout_t        time;      //< Timeout in RTimer format
17 } os_timeout_t;
```

Program 4. *Timeout service public and private interface*

The only mandatory parameter is the callback, defined as:

```
typedef void(*os_timeout_cb_f)(void * p1, uint32_t p2)                    (4)
```

where p1 and p2 are the parameters given in the parameter structure.

When a timeout is registered, the kernel sets the earliest timeout to the timer HAL compare channel. When the ISR is triggered all clients with this timeout value are notified via the callback, the resources are freed, and a new timeout is set to the timer HAL. Timeouts can be registered and aborted dynamically, and a task can have multiple timeouts reserved.

6.3.3 Real-time service

The kernel part of the real-time service is a very lightweight abstraction for the real-time event HAL service. The kernel part is used for claiming ownership of the HAL block, and checking task privileges when trying to access the real-time event service.

User set a real-time event by calling:

```
bool Os_setRtEvent(os_rtevt_cb_f user_callback, uint32_t timestamp)       (5)
```

Where user_callback is an optional callback invoked when the event time occurs, and timestamp is the time of the event as an absolute system timestamp.

If no callback is required, users can poll for the event, or wait for the event to occur. This wait state enters a low power mode and blocks task execution until the event has passed. This wait state cannot be timed out, as reserving a timeout generates unnecessary overhead.

6.4 Energy management

The system energy management is handled by the kernel and kernel HAL by keeping record of reserved resources and determining the optimal low power mode when a system sleep period is requested. Keeping record of hardware resource utilization is necessary to prevent the system from entering certain sleep states where the High Frequency Clock (HFCLK) is disabled. All peripherals in the HFCLK domain need this clock in order to work, and entering the deepest sleep state automatically disables this clock source to preserve energy [11].

The bookkeeping part is done with a LUT keeping reference count for all the supported hardware peripherals. Users can request activation or de-activation of peripherals and each request modifies the reference count and controls the peripherals clock accordingly. When a low power mode is requested, the reference counters are used to determine the lowest possible sleep state and if the HFCLK can be disabled.

When requesting operations on a certain peripheral it is accessed by a symbolic name defined in the public interface of the energy management module. To speed optimize the access, this symbolic name is also the reference count LUT index. Users can also explicitly disable the deep sleep states on a client basis by setting values in a bitmask inside the energy management module itself. If any client bit is set, entering the deepest possible sleep state is prohibited.

How the energy management is actually implemented is entirely platform dependent, so the operating system kernel itself does not implement any peripheral control or how the optimal power states are determined. However, the kernel is in full control of deciding when the system enters sleep and which events can wake the system up.

6.5 Message queues

Program listing 5 introduces the private mailbox of a message queue for the kernel. The message queues are implemented as a single linked list, with ownership information. Not all tasks are required to have a message queue, and each queue must be allocated by tasks if they want to receive messages. The owner information is used to notify the owner of the queue of a received message by the kernel. The scheduler is also notified, in case the system is running idle task to schedule the receiving task immediately.

```
1  // Private container for message queues
2  typedef struct
3  {
4      sl_list_head_t      queue;   //< The queue mechanism (linked list)
5      os_thread_id_t      owner;   //< Owner of the queue
6      uint8_t             padd[2]; //< Padding for EABI alignment
7  } os_msg_queue_t;
8
9  // Declaration of a mailbox handle
10 typedef void *          os_msg_queue_handle_t;
11
12 // Declaration of os_msg_t
13 typedef sl_list_t       os_msg_t;
14
15 // Declaration of an example message envelope, that must inherit os_msg_t
16 typedef struct
17 {
18     os_msg_t            header;  //< Inherited header
19     uint8_t             id;      //< Message identifier
20     uint8_t             padd[3]; //< For EABI alignment
21     void *              msg;     //< Message content
22 } os_example_message_t;
23
24 // Example of a commonly agreed mailbox name for a target application
25 #define PID_APP         app_queue
26
27 // Defined as weak, if application does not implement a queue
28 os_msg_queue_handle_t   app_queue __attribute((weak));
```

Program 5. *Message queue private mailbox, and public declaration*

When sending messages the sender must allocate memory for the message and know the mailbox name of the target. The mailbox names are globally pre-assigned for different layers and exposed via a layer of abstraction. Program listing 5 presents an example implementation for a mailbox name, and a prototype envelope. This global declaration of mailbox names gives the benefit that a message can be delivered directly to the mailbox, and the operating system does not have to implement a procedure to dispatch messages.

Sending and receiving are both O(1) time operations [12]; sending is a push back operation, while receiving is pop front, both of which just update list linkage. This also ensures that the messages are received in order. Other supported operations include *flushing* a queue, which removes all elements, and *waiting* which is a type of synchronization. Performing a wait operation on a queue blocks the task and enters a low power state, until a message is received or the operation is timed out.

6.6 Interrupt priorities

Determining interrupt priorities is necessary to ensure that the most critical tasks will always get execution priority and will never be preempted, unless a higher priority task needs servicing. The highest priority interrupt source must always be the real-time event service. Other critical sources might be serial interfaces, or edge detection on GPIO pins for sensor state monitoring e.g. when new data is ready.

Table 4 presents an example configuration of interrupt priority levels on a Cortex-M4 platform. Priority levels are inverted, meaning lower number means higher priority.

Table 4.Example of NVIC interrupt priorities

Task	Interrupt source	Priority level
Real-time event execution	High frequency timer	0
Real-time event pre-trigger	Real-Time Clock (RTC)	0
Timeout handling	Real-Time Clock (RTC)	1
Clock source ready interrupt	Energy management	2
USART interrupts	Asynchronous serial interface	2
SPI interrupts	Synchronous serial interface	2
GPIO edge detect	Sensor data ready pins etc.	3

6.7 System calls

Table 5 presents the task control API and Table 6 presents the system service call API functions. Function parameters are omitted from the declarations. Relevant parameters are mentioned in the description field.

Table 5.Task control and kernel internal service calls

Function name	Description
void Os_schedulerStart()	Start the system scheduler. Function does not return. New tasks cannot be introduced once the scheduler is started.
os_thread_id_t Os_createTask()	Create a new task.
bool Os_updateTaskPriority()	Attempt to update task priority. Returns false if fails.
bool Os_updateTaskDuration()	Attempt to update task duration information for the scheduler. Returns false if fails.
bool Os_updateTaskStart()	Attempt to update task start time in system internal timestamp for the scheduler. Returns false if fails.
os_task_t * Os_findTask	Find a task assigned with a PID. Kernel internal system call.
void Os_signalTaskTimeout()	Signal task of timeout. Kernel internal system call.
void Os_signalTaskWakeup()	Signal task of event that moves it to ready state. Kernel internal system call.
void Os_clearStatus()	Clears the tasks status field, timeout and wakeup. Kernel internal system call.
void Os_schedulerNotifyEvent()	Notify the scheduler of an event. Kernel internal system call.
os_task_t * Os_createIdleTask	Spawn the idle task. Kernel internal system call.
os_dsr_handle_t Os_addDsr	Add a deferred service routine to the scheduler. Kernel internal system call.
void Os_removeDsr()	Remove deferred service routine from the scheduler. Kernel internal system call.

Table 6.Operating system service calls

Function name	Description
void Em_requestPeripheralActive()	Request peripheral activation from energy management. Parameter name determines which block is activated.
void Em_requestPeripheralDeactive()	Request peripheral de-activation from energy management. Parameter name determines which block is de-activated.
uint32_t RTimer_getUs()	Request system microsecond timestamp. The value returned is the entry time for function.
bool Os_setRtEvent()	Setup a real-time event. Event time in system time and optional callback are required. Returns false if operation fails.
void Os_waitRtEvent()	Wait until a previously set real-time event occurs. Returns immediately if real-time event not allocated. System enters a low power mode for the wait duration.
os_timeout_handle_t Os_createTimeout()	Create an alarm timeout. Mandatory timeout value required. Timeout is calculated from current system time at function entry. Returns handle to event.
void Os_abortTimeout()	Abort a timeout. Requires handle to existing timeout. Behaves as if the service call has succeeded if timeout has passed or resource is abandoned.
os_event_handle_t Os_createEvent()	Create event. Handle to event is returned and event must be accessed via handle.
void Os_destroyEvent()	Destroy event (frees up resource for others).
bool Os_waitEvent()	Wait for an event to be signaled. Mandatory timeout parameter, which can be used to poll signal or wait forever.
void Os_signalEvent()	Signal an event. Called for example from interrupt handlers.
void * Os_memAlloc()	Allocate n bytes of dynamic memory.
void Os_memFree()	Free allocated memory.
os_msg_queue_handle_t Os_msgCreateQueue()	Create a message queue. Returns handle to queue or NULL if resource cannot be allocated.
void Os_msgFlushQueue()	Flushes a message queue by freeing each element.
bool Os_msgSend()	Send message to queue. Requires that memory for message has been pre-allocated. Returns false if resource (the queue) does not exist.
os_msg_t * Os_msgReceive()	Fetch a single message from a message queue. Returns NULL if the queue does not exist.
bool Os_msgQueueWait()	Wait until a queue is signaled (a new message is received). Blocks execution and enters low power mode.

6.8 List of files and directories

Table 7 presents files and folders that implement WPC-OS. Only the relevant HAL parts are listed here.

Table 7.List of files and folders of WPC-OS implementation

File	Description
/mcu/	Root directory for HAL files
/mcu/interrupt.h	Interrupt handling declarations
/mcu/hal_api/	Root directory for HAL API
/mcu/hal_api/rtimer.h	System timer HAL declarations
/mcu/hal_api/energy_management.h	Energy management HAL declarations
/mcu/nrf52/rtimer.c	System timer implementation for nRF52
/mcu/nrf52/energy_management.c	Energy management implementation for nRF52
/mcu/nrf52/interrupt.c	Interrupt handling implementation
/mcu/nrf52/rtimer/rtimer.c	System timer implementation for nRF52
/os/	Root directory for the WPC-OS kernel
/os/os.h	WPC-OS public service call API declarations
/os/events.h	Event API declarations
/os/events.c	Event API implementation
/os/interrupt.h	Deferred service routine API declarations
/os/interrupt.c	Deferred service routine API implementation
/os/kernel.h	Kernel public API declarations
/os/kernel.c	Kernel API implementation
/os/mem.h	Dynamic memory API declarations
/os/mem.c	Dynamic memory API implementation
/os/msg.h	IPC message API declarations
/os/msg.c	IPC message API implementation
/os/os_private.h	Kernel private API declarations
/os/os_rtevent.h	Real-time event API declarations
/os/os_rtevent.c	Real-time event API implementation
/os/os_scheduler.h	Scheduler public API declarations
/os/os_scheduler.c	Scheduler implementation
/os/timeout.h	Timeout API declarations
/os/timeout.c	Timeout API implementation
/util/	Root directory for generic utilities
/util/sl_list.h	Single linked list API declarations
/util/sl_list.c	Single linked list API implementation

7. EVALUATION AND MEASUREMENTS

In this chapter WPC-OS functionality is measured and the results are evaluated against the requirements in Table 3. The tests include simple feature verification tests and finally running the full WPC stack with an application.

7.1 Measurement setup

Measurements are performed on an nRF52832 PCA10040 Development Kit-board connected to a Saleae Logic16 [22] logic analyzer with GPIOs. Figure 15 shows the basic output of the logic analyzers window. The measured pins are stacked on top of each other and each measurement pin is given a name describing what phenomenon it is measuring. This setup is used to measure timings, MCU activity, context switch delays and task execution. The software also supports exporting csv-formatted data that will be analyzed statistically.

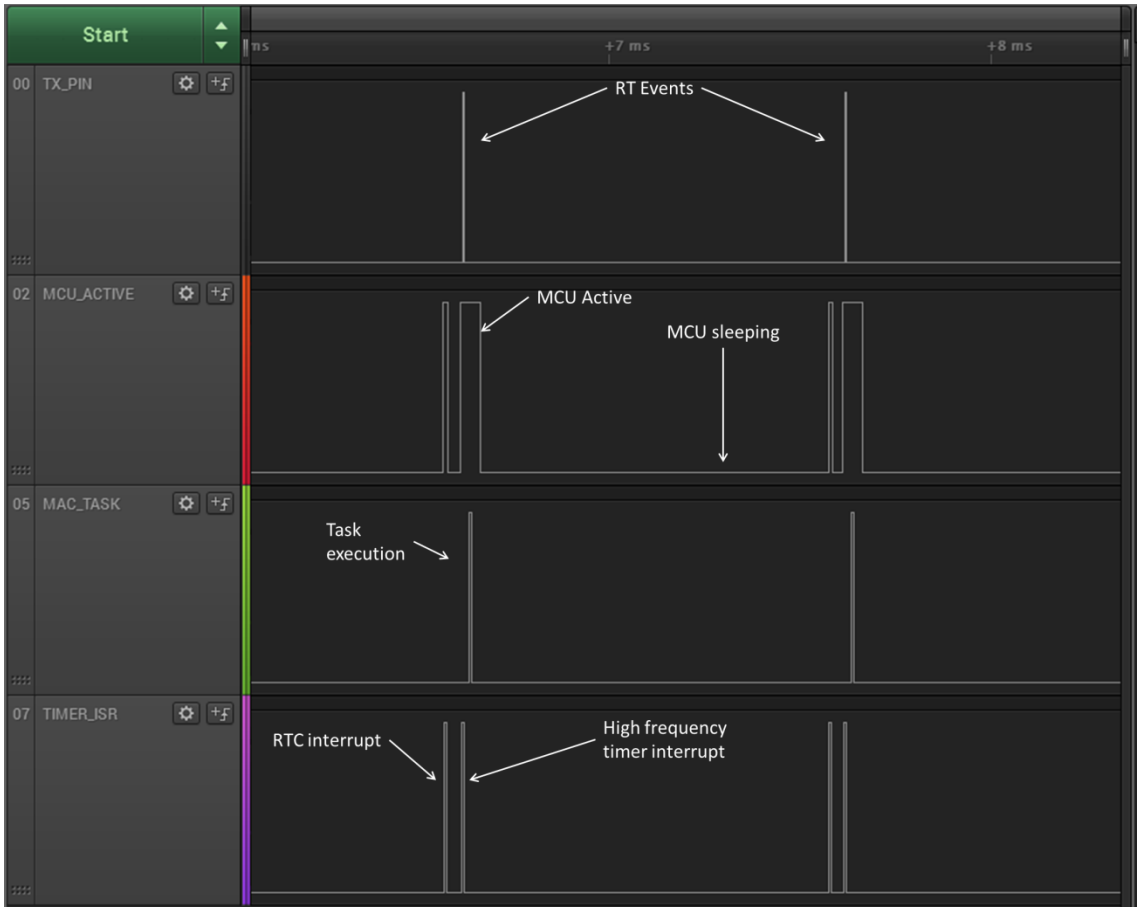

Figure 15.Measuring instrument screen

7.2 Real-time events

Measuring real-time event accuracy and predictability will prove that WPC-OS provides a fully deterministic and microsecond accurate timing mechanism with its real-time event service. Timing characteristics will be measured with a 1ms, 10ms and 100ms interval and the results analyzed statistically. The interesting measurement points are the 0 → 1 transition times of the first channel, named TX_PIN.

The statistical analysis results are in Table 8, where E(X) is the expected value of event interval, N is the number of samples obtained, AVG(X) is the measured average, VAR(X) is the variance and DEV(X) is the standard deviation over N samples. The most interesting numbers however are the MIN(X) and MAX(X) numbers showing that the timings are glitch free, and AVG(X) which indicates a positive bias in the timings with longer periods. This bias is due to a constant DC-bias and a temperature coefficient in the external 32.768 kHz crystals output frequency [23].

Table 8.Real-time event timing statistics

E(X) (us)	N (samples)	AVG(X) (us)	VAR(X) (us)	DEV(X) (us)	MIN(X) (us)	MAX(X) (us)
1000	2348	1000.50	0.57	0.75	999.20	1002.72
10000	6458	10001.21	0.67	0.82	9999.68	10003.28
100000	1687	100010.11	0.50	0.70	100008.80	100012.30

Predictability and consistent accuracy is ensured by running the real-time event task in the highest priority interrupt context, meaning all other tasks and interrupts will be preempted and no other operation can interfere with this timing.

7.3 Energy efficiency and energy management

Proving energy efficiency is difficult without a reference, but some statistics from the system can be gathered and cross referencing the statistics with theoretical values provides insight on system performance. For this measurement the system was running one real-time priority task with 1ms interval and energy saving was handled by the idle task.

Figure 16 shows what happens in the system when a real-time event occurs and the real-time priority task is signaled. At first, the system is sleeping and running the idle task. The first interrupt and MCU active states are the RTC interrupt handler waking the system from the deepest sleep state to trigger a HFTIMER interrupt. The second interrupt is the actual real-time event run from the HFTIMER interrupt handler. After the event, the real-time task is executed and idle task is re-entered.

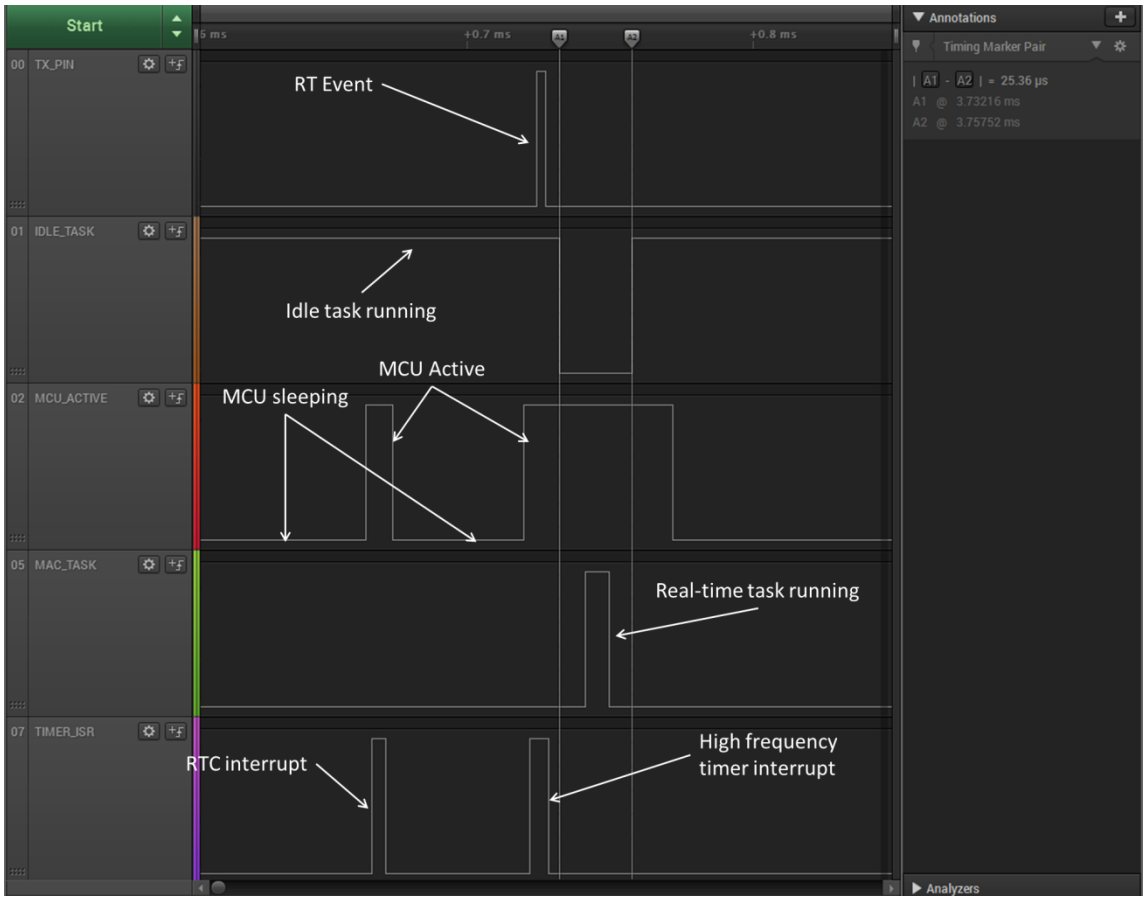

Figure 16.Task wake-up by real-time event

Table 9 presents timings that are required to estimate operating system performance.

Table 9.Measured timings from energy efficiency setup

Symbol	Explanation	Value (us)
T_{rttask}	Real-time task duration	8.4
$T_{idle2idle}$	Average time from idle task end to re-enter idle task	25.4
$T_{rtcmargin}$	RTC pre-margin to set up HFTIMER for real-time event	57.7
T_{active}	MCU active time within 1ms period	52.3
T_{sleep}	MCU sleep time within 1ms period	947.7
T_{ds}	Deep sleep period over T_{sleep}	890.0

The system states can be defined as:

- **Deep sleep:** RTC clock running, High Frequency Clock (HFCLK) off, CPU off
- **Idle sleep:** HFCLK running, CPU off
- **Run mode:** HFCLK running, CPU on

7.3.1 Deep sleep utilization

The nRF52832 CPU consumes 2.2 uA while in deep sleep waiting for RTC wake-up and 60 uA in idle sleep [21]. This means the CPU is using 27 times more energy when system sleep periods do not utilize deep sleep, thus maximizing deep sleep utilization is essential for smart energy management. In Figure 16 the system is in deep sleep until the RTC interrupt. From this time until the HFTIMER interrupt, the system is in idle sleep (because the HFCLK is needed for the HFTIMER).

To measure automatic energy management efficiency, estimating the proportion between deep sleep and idle sleep utilization can be used. The proportion of sleep states for a 1ms period can be calculated roughly as:

$$Ratio_{ds2is} = \frac{T_{ds}}{T_{sleep} - T_{ds}} = \frac{890.0}{947.7 - 890.0} = 15.4 \quad (6)$$

This means the system uses deep sleep over 15.4 times more than idle sleep during a 1ms period. As the RTC pre-margin time is constant, this ratio will decrease with shorter sleep periods and increase with longer sleep periods.

7.3.2 Idle task utilization

While in run mode, the nRF52832 uses 3.712mA, which is over 1000 times more than in deep sleep mode. Thus maximizing idle task utilization is necessary for energy saving. Unnecessary wake-ups must be avoided, so the system must stay in idle mode when it has nothing to do. The system must also run in a *tickless* manner, meaning there is no periodic timer waking the system up.

By subtracting the idle task turnaround from the 1ms period a theoretical idle/active period proportion can be calculated and cross referenced with a statistical analysis of the IDLE_TASK pin duty cycle. Over a sampling period of 130 seconds the measured utilization was 97.455% while the theoretical idle task utilization should be 97.460%. The values match, meaning the system is not waking up unnecessarily.

7.3.3 Context switching delay

From Table 9 the average context switching delay can also be calculated as

$$T_{cx} = \frac{T_{idle2idle} - T_{rttask}}{2} = \frac{25.4 - 8.4}{2} us = 8.5us \quad (7)$$

The context switching delay is homogenous, regardless of task priority. As all tasks are run to completion context switch times when a lower priority task is running depends entirely on when the task decides to yield, but the operating system overhead stays constant.

7.4 Memory footprint

The WPC-OS kernel requires 110 bytes of data memory and 4977 bytes of program memory. When compiled for the nRF52832 platform these figures amount to 0.2% data memory and 0.9% program memory usage respectively. These figures are the static memory requirements of the kernel and kernel HAL.

The maximum amount of tasks is build time configurable and this affects data memory consumption. One task requires a TCB which is 24 bytes each but as all tasks share the same stack memory increasing the amount of tasks does not increase memory consumption any further. Other dynamically configurable OS components include events (4 B), timeouts (16 B), message queues (16 B) and dynamic memory which requires 24 B for each different slab size. Program memory consumption does not change with configurable parameters.

WPC requires the following tasks; real-time priority MAC, high priority routing and stack management with one task each, a low priority stack maintenance task and four medium priority stack support applications. User applications require a task for running the application and an application support task, both of which are low priority. This amounts to a minimum of 10 tasks. Each task is assigned a single event, timeout and message queue. With 3 different slab sizes the data memory consumption is 663 bytes which amounts to 1% RAM usage. In addition a stack must be allocated, which is 1 kB. With preemptive solutions where each task requires its own stack the memory consumption exceeds 10 kB with stack memory allocation alone, not counting other operating system objects and their memory needs.

The WPC-OS memory usage is detailed in Table 10. Data memory usage is calculated with the minimum configuration.

Table 10.WPC-OS memory footprint with WPC

Module name	Program memory usage (bytes)	Data memory usage (bytes)
Kernel	692	256
Events	386	36
Real-time events	168	6
Scheduler	500	4
Timeouts	381	16
Timer HAL	1506	28
Message queues	193	160
Memory management	401	113
Energy management	592	44
Single linked list	158	0
Total	4977	663

7.4.1 Memory footprint comparison

Program memory usage comparison to other operating systems would require building each operating system with the same compiler for the same platform and thus is not feasible. Consequently, program memory usage efficiency is only estimated. Data memory usage comparison against preemptive kernels is not feasible due to the stack memory requirements alone. Therefore, comparison should be done with another event-driven OS.

Contiki is selected for comparison as it is event-driven and written in ANSI C language. In the comparison example Contiki was built for TI CC2430 SoC for Sensinode platform [24]. The CC2430 SoC implements an 8051 [25] CPU. The build includes an IPV6 over Low power Wireless Personal Area Networks (6LoWPAN) stack with a sensor application.

Comparing program memory usage is not possible directly due to different instruction sets between the 8-bit 8051 and 32-bit ARM CPUs, but an estimate can be made. For comparison, the Core and Timer modules were selected from the Contiki example build. The example uses a stack of 223 B, which is subtracted from core data memory usage.

The Contiki example build requires 13346 B of program memory while the WPC-OS build requires 4977 B. The Thumb-2 instruction set for ARM provides 23% better code density compared to IA32 [26], but this is application dependent. Assuming a code compression rate of 50% in favor for the ARM, WPC-OS still uses less program memory.

As for data memory usage, the Contiki core uses 158 B [24], while WPC-OS uses 110 B. However, these figures do not include TCB allocations. To clarify terms, a Contiki process [27] will be defined as TCB, as it is the control block of the event-driven part. A Contiki TCB requires 18 B of data memory, while a WPC-OS TCB uses 24 B.

As a conclusion, WPC-OS has lower program memory usage and similar data memory usage when compared to Contiki.

7.5 Example task

Program listing 6 presents an example real-time application for WPC-OS. This is the application used for real-time accuracy and energy management measurements. A call to *spawn_rt_task()* initializes the information needed by the kernel, creates the real-time task and assigns a message queue (symbolically named PID_MAC) for the task. Finally, this function sends a message to PID_MAC, which automatically sets the task event informing the scheduler that the task has work to do.

```
1  // Interval for RT events
2  #define RT_EVT_TIME     HAL_USEC_TO_UPERIOD(1000u)
3
4  os_msg_queue_handle_t   mac_queue;      // Message queue for task
5  static uint32_t         mac_boundary;   // Real-time event boundary
6  #define PID_MAC         mac_queue
7
8  static void             rt_task(void)
9  {
10     // Perform RT task
11     GPIO_ENTER_TX();
12     // Wake MAC up again
13     Os_msgSend(PID_MAC, NULL);
14     GPIO_EXIT_TX();
15 }
16
17 static uint32_t         mac_task(void)
18 {
19     // Spawn RT event
20     static bool rt_time_valid = false;
21     GPIO_ENTER_MAC();
22     if(!rt_time_valid)
23     {
24         // Obtain start time
25         mac_boundary = RTimer_getUs();
26         rt_time_valid = true;
27     }
28     // To prevent drift move the rt event boundary forward instead of
29     // using current system time here
30     mac_boundary += RT_EVT_TIME;
31     // Setup RT event with callback
32     Os_setRtEvent(rt_task, mac_boundary);
33     GPIO_EXIT_MAC();
34     // This task does not follow a timetable (pure async)
35     return OS_NO_TIMETABLE;
36 }
37
38 static void             spawn_rt_task(void)
39 {
40     // Create example task
41     os_task_info_t mac;
42     os_thread_id_t id;
43     mac.task_name = "MAC_TASK";
44     mac.priority = OS_PRIO_REAL_TIME;
45     mac.func = mac_task;
46     mac.interval = 0; // No interval as task is pure async
47     mac.duration = 20; // Measured duration is 8.4us. Some margin added.
48     // Record own task id
49     id = Os_createTask(&mac);
50     mac_queue = Os_msgCreateQueue(id);
51     // Send message to self to wake-up immediately
52     Os_msgSend(PID_MAC, NULL);
53 }
```

Program 6. *Creating an example task with real-time priority*

The *mac_task()* function is the context in which the real-time task is executed. The execution is indicated by a GPIO pin symbolically named GPIO_ENTER/EXIT_MAC. On first entry, the task stores the system timestamp and triggers a real-time event, with a callback to *rt_task()* 1000 us from this time and exits. On subsequent entries the stored time is moved forward by 1000 us and a new event is triggered. Obtaining the system timestamp only once removes the timing delay caused by the scheduler context switch.

The callback function is executed in the HFTIMER context. It toggles another pin called GPIO_ENTER/EXIT_TX and sends a message to PID_MAC, waking up the task. The moment indicated by GPIO_ENTER_TX was used for measuring real-time event accuracy.

7.6 Running WPC with WPC-OS

Running WPC with WPC-OS was tested by implementing a UART command interpreter application on the WPC node and a command generator on a PC. This test implements the minimum WPC configuration with 10 tasks and idle task. The test software was built on the nRF52832 platform, which has a total of 512 kB of program memory and 64 kB of data memory. Table 11 shows the memory consumption of each layer in this example.

The embedded application runs in an asynchronous manner meaning it notifies the WPC-OS kernel and gets scheduled whenever a command from the PC is received. UART interrupts are used to send and receive the serial data on the application. Communication between WPC and the UART application is done via IPC messages. Figure 17 illustrates the test setup.

The PC application sends 10 packets from one node and the other node receives this data and pushes it to the PC. Sent and received packets were printed on terminal software on the PC. Figure 18 shows the terminal screen from both nodes with a sequence number. All packets were sent and received successfully throughout the entire chain.

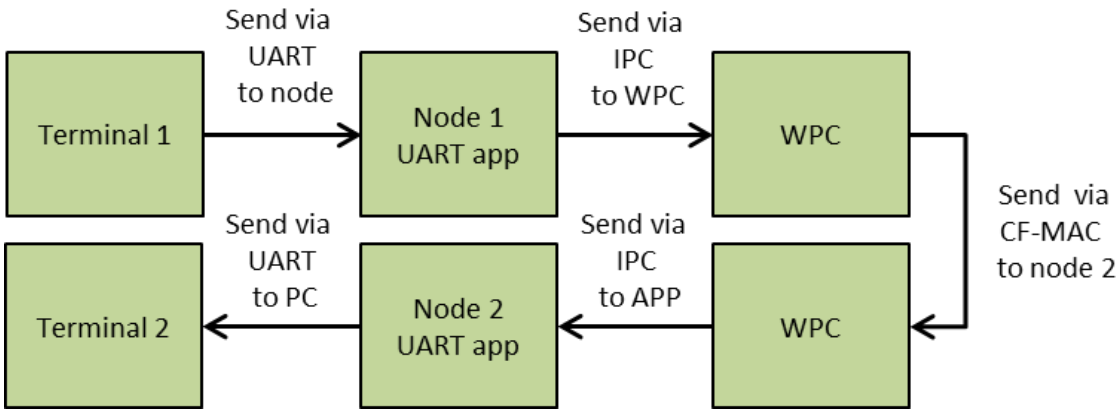

Figure 17.WPC with WPC-OS test setup

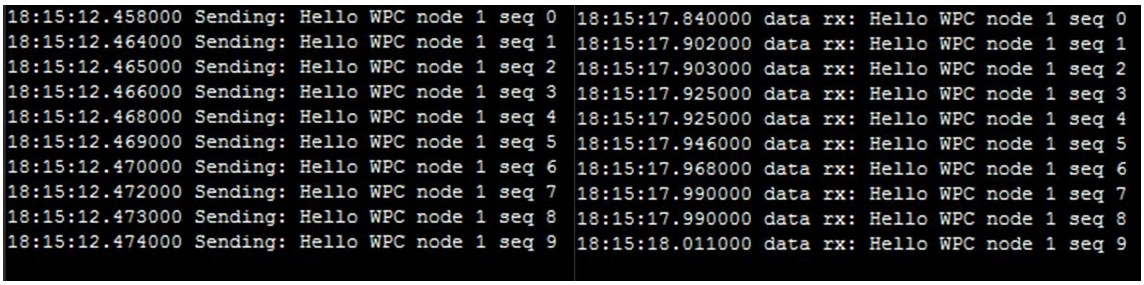

Figure 18.Terminal output from both nodes

Table 11.Memory usage of WPC-OS with WPC example

Layer	Program memory usage (bytes)	Data memory usage (bytes)
WPC-OS	4977	663
WPC	78572	5979
WPC support tasks	4026	1183
Application support tasks	5578	162
Serial port application	10048	6256
Total	103201	14243
Total used (%)	19.68	21.73

7.7 Evaluation and measurement results

Table 12 gathers requirements, measurement targets and results. WPC-OS fulfills all WPC requirements. WPC-OS provides an accurate and deterministic real-time event service. It supports running multiple tasks while maintaining energy efficient operation and a small memory footprint.

Table 12.Evaluation and measurement results

Requirement	Target	Result	Target met?
Real-time event accuracy	Below 1 XTAL tick (30.5us)	Under 1 us	Yes
Energy-efficiency	CPU active only when necessary	CPU is active only when necessary	Yes
Tickless	Remove unnecessary wake-ups	No unnecessary wake-ups	Yes
Optimized memory footprint	Usages: Program memory < 10kB Data memory < 1kB	Usages: Program memory < 5kB Data memory < 500B	Yes
Automatic energy management	Implement idle task and a centralized energy management module.	Idle task and energy management implemented	Yes
Ability to run multiple tasks	Minimum amount of tasks: 10 tasks + idle task	The minimum WPC configuration works	Yes

8. CONCLUSIONS

IoT interconnects embedded devices. This is traditionally done by using broadband and mobile Internet, which require one connection per device. The WPC protocol stack removes this requirement by connecting the IoT devices directly to each other. An IoT device with WPC is intended to be battery powered and is expected to work for years without battery change. Thus, WPC is designed to work on resource constrained embedded hardware, with limited memory and processing power. These constraints complicate application design and the solution is provided by an RTOS kernel.

This thesis presented an RTOS kernel called WPC-OS. The kernel design and implementation were presented. WPC-OS timing accuracy, energy efficiency and memory footprint were evaluated.

The WPC-OS kernel was designed for the sole purpose of running WPC with customer applications. WPC places requirements on RTOS design in the form of resource constrained platforms and timing critical CF-MAC operation. WPC-OS solved all design requirements WPC imposes on RTOS design. Its viability was confirmed by implementing it on a nRF52832 platform and running it with a serial port application and WPC sending data between two nodes.

The WPC-OS kernel uses under 5kB of program memory and 350 B of data memory. Compared to Contiki, the program memory usage is less while data memory usage is similar. The WPC-OS kernel provides a deterministic real-time event service with less than 1 us timing error. It simplifies customer application design by providing multitask capability and centralized hardware access. It provides a novel and lightweight way of running tasks with periodic work with its timetable scheduling, which uses task durations to determine the next task. It improves WPC reliability when running such applications by making it impossible to interfere with protocol timings. These features are expected to cut time to market of customer products with WPC.

The future development of WPC-OS should include investigating the feasibility of using co-routines [17] or a hybrid scheduler [41] for task scheduling. They provide preemptive multitask scheduling combined with the light-weightiness of cooperative scheduling. Other desired features might include run-time configuration of operating system resources (such as tasks) and implementing memory area protection (for safer task handling). However, these new features must not compromise the light-weightiness

of the kernel. Minimizing the operating system processing overhead and memory consumption must take priority over programming convenience.

REFERENCES

[1] Telecommunication Standardization Sector of International Telecommunication Union, Overview of the Internet of things, [ONLINE] Available: http://handle.itu.int/11.1002/1000/11559, 2012

[2] Dake Liu and C. Svensson, Power consumption estimation in CMOS VLSI chips, in *IEEE Journal of Solid-State Circuits*, vol. 29, no. 6, pp. 663-670, 1994.

[3] Francisco Martinez-Sanches, Time to Market and Impatient Customers, *Bulletin on Economic Research 65:2*, 2013.

[4] Qing Li, Carolyn Yao, *Real-Time Concepts for Embedded Systems*, CMP Books, 1st edition, 2003.

[5] William Stallings, *Operating Systems: Internal and Design Principles*, 6th edition, Pearson, 2008.

[6] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, *Operating System Concepts*, 7th edition, John Wiley & Sons Ltd, 2005.

[7] Andrew S. Tanenbaum, Herbert Bos, *Modern Operating Systems*, 4th edition, Pearson, 2015.

[8] Wirepas homepage, Technology [ONLINE] Available: http://www.wirepas.com/, 2017-05-09.

[9] Mauri Kuorilehto, Mikko Kohvakka, Jukka Suhonen, Panu Hämäläinen, Marko Hännikäinen, Timo D. Hämäläinen, *Ultra-Low Energy Wireless Sensor Networks in Practice – Theory, Realization and Deployment.* John Wiley & Sons Ltd, 2007

[10] Jason Hill, Rober Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister, System Architecture Directions for Networked Sensors, *Proc. 9th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00),* Cambridge, 2000

[11] Advanced RISC Machines, Cortex-M3 technical reference manual [ONLINE] Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337e/DDI0337E_cortex_m3_r1p1_trm.pdf, r1p1, 2017-03-22

[12] Thomas H. Cormen, Charles E. Leiserson, Donald L. Rivest, Clifford Stein, *Introduction to algorithms,* 3rd edition, Massachusetts Institute of Technology, 2009

[13] A. Dunkels, B. Gronvall and T. Voigt, Contiki - a lightweight and flexible operating system for tiny networked sensors, *29th Annual IEEE International Conference on Local Computer Networks*, pp. 455-462, 2004

[14] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, Muneeb Ali, Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems, *SenSys '06 Proceedings of the 4th international conference on Embedded networked sensor systems Boulder, Colorado, USA*, pp 29-42, 2006

[15] TinyOS Wiki, Wikipedia & FAQ [ONLINE]
Available: http://tinyos.stanford.edu/tinyos-wiki, 2017-03-27

[16] Philip Levis, TinyOS Programming [ONLINE]
Available: http://csl.stanford.edu/~pal/pubs/tinyos-programming.pdf, 2017-03-27

[17] FreeRTOS homepage, Coroutines, About, Features, Software Timers [ONLINE]
Available: http://www.freertos.org/, 2017-03-28

[18] Micrium RTOS, Micro-Controller Operating Systems (µC/OS) [ONLINE]
Available: https://www.micrium.com/rtos/, 2017-03-28

[19] American National Standard Institute, Programming Languages – c [ONLINE]
Available: http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf, 2017-04-27

[20] GNU, GCC, the GNU Compiler Collection, [ONLINE]
Available: https://gcc.gnu.org/, 2017-04-27

[21] Nordic Semiconductor, nRF52832 Product Specification [ONLINE]
Available: http://infocenter.nordicsemi.com/pdf/nRF52832_PS_v1.1.pdf, 2017-03-22

[22] Saleae Logic 16, The Original Logic 16 Logic Analyzer [ONLINE]
Available: https://www.saleae.com/originallogic16, 2017-04-12

[23] Seiko Epson Corporation, FC-135R kHz Range Crystal Unit [ONLINE]
Available: http://www.mouser.com/ds/2/137/1699449-465441.pdf, 2017-04-27

[24] G. Oikonomou and I. Phillips, Experiences from porting the Contiki operating system to a popular hardware platform, *2011 International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS)*, Barcelona, pp. 1-6, 2011

[25] Intel Corporation, MCS 51 Microcontroller Family User's Manual, 1994

[26] M. Breternitz, H. Hum, R. Peri, J. Pickett and Y. Wu, Enhanced code density of embedded CISC processors with echo technology, *2005 Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'05)*, Jersey City, NJ, USA, pp. 160-165, 2005

[27] Contiki git repository. File: Contiki/core/sys/process.h [ONLINE] Available: https://github.com/contiki-os/Contiki, 2017-04-19

[28] Advanced RISC Machines, Cortex-A9 technical reference manual [ONLINE] Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388f/DDI0388F_cortex_a9_r2p2_trm.pdf, r2p2, 2017-03-22

[29] Advanced RISC Machines, ARM Architecture Reference Manual [ONLINE] Available: https://silver.arm.com/download/download.tm?pv=1874087, 2017-03-22

[30] Advanced RISC Machines, ARM Compiler toolchain: Developing Software for ARM processors, Version 5.02, 2010-1012

[31] Libelium, 50 Sensor Applications for a Smarter World [ONLINE] Available: http://www.libelium.com/sections_files/applications/Download/Libelium_50_sensor_applications.pdf, 2017-04-26

[32] Juha Korhonen, *Introduction to 3G Mobile Communications*, 2nd edition, Artech House, 2003

[33] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis, *Integrating concurrency control and energy management in device drivers, in Proc. ACM SOSP*, 2007.

[34] Texas Instruments, CC2650 Reference Manual [ONLINE] Available: http://www.ti.com/lit/ug/swcu117g/swcu117g.pdf, 2017-05-02

[35] Silicon Laboratories, EZR32LG Wireless MCUs datasheet [ONLINE] Available: https://www.silabs.com/documents/public/data-sheets/EZR32LG330_DataSheet.pdf, 2017-05-02

[36] Silicon Laboratories, EFR32MG1 Mighty Gecko ZigBee & Thread SoC Family Data Sheet [ONLINE] Available: https://www.silabs.com/documents/public/data-sheets/efr32mg1-datasheet.pdf, 2017-05-02

[37] ST Microelectronics, STM32F-series MCU datasheet [ONLINE] Available: www.st.com/resource/en/datasheet/stm32f405og.pdf, 2017-05-02

[38] ST Microelectronics, Ultra-low-power STM32L0x1 advanced ARM-based 32-bit MCUs, Reference manual [ONLINE] Available: www.st.com/resource/en/reference_manual/dm00108282.pdf, 2017-05-02

[39] Nordic Semiconductor, nRF51822 Reference Manual [ONLINE] Available: http://infocenter.nordicsemi.com/pdf/nRF51_RM_v3.0.pdf, 2017-05-02

[40] M. Elkhodr, S. Shahrestani and H. Cheung, The Internet of Things: Vision & challenges, *IEEE 2013 Tencon - Spring*, Sydney, NSW, pp. 218-222, 2013

[41] T. Laukkarinen, V. A. Kaseva, J. Suhonen, T. D. Hamalainen and M. Hannikainen, "HybridKernel: Preemptive kernel with event-driven extension for resource constrained wireless sensor networks," *2009 IEEE Workshop on Signal Processing Systems*, Tampere, pp. 161-166, 2009

[42] Advanced RISC Machines, Cortex-M4 technical reference manual [ONLINE] Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0439b/DDI0439B_cortex_m4_r0p0_trm.pdf, 2017-05-09

[43] Advanced RISC Machines, Cortex-M3 Embedded Software Development [ONLINE] Available: http://infocenter.arm.com/help/topic/com.arm.doc.dai0179b/AppsNote179.pdf, 2017-05-11